

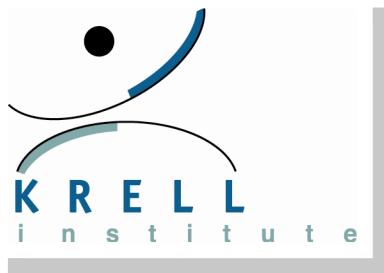
Open | SpeedShop™

How to Analyze the Performance of Parallel Codes 101 *A case study with Open|SpeedShop*

SciDAC 2011 Tutorial

Jim Galarowicz, Krell Institute

Mahesh Rajan, SNL



❖ Performance Analysis is becoming more important

- Complex architectures
- Complex applications
- Mapping applications onto architectures

❖ Often hard to know where to start

- Which experiments to run first?
- How to plan follow-on experiments?
- What kind of problems can be explored?
- How to interpret the data?

❖ Provide basic guidance on ...

- How to understand the performance of a code?
- How to answer basic performance questions?
- How to plan performance experiments?

❖ Basics on Open|SpeedShop

- Introduction into one possible tool solution
- Basic usage instructions
- Pointers to additional documentation

❖ Provide you with the ability to ...

- Run these experiments on your own code
- Provide starting point for performance optimizations

❖ Open Source Performance Analysis Tool Framework

- Most common performance analysis steps ***all in one tool***
- ***Extensible*** by plugins for data collection and representation
- Gathers and displays several types of performance information

❖ Flexible and Easy to use

- User access through ***GUI, Command Line, Python Scripting, and convenience scripts.***

❖ Several Instrumentation Options

- All work on ***unmodified application binaries***
- ***Offline*** and ***online data collection / attach*** to running codes

❖ Supports a wide range of systems

- Extensively used and tested on a variety of ***Linux clusters***
- New: ***Cray XT/XE*** and ***Blue Gene/P*** support

❖ Let's keep this interactive

- Feel free to ask questions as we go along
- Online demos as we go along
- Ask if you would like to see anything specific

❖ We are interested in feedback on O|SS

- What is good?
- What should be done differently?
- What is missing?

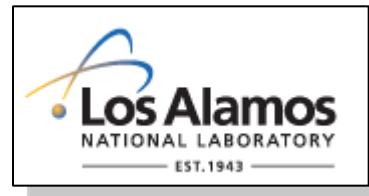
❖ Please report bugs and incompatibilities

Presenters



- ❖ **Jim Galarowicz, Krell**
- ❖ **Mahesh Rajan, SNL**
- ❖ **Larger team**

- David Montoya, LANL
- Donald Maghrak, Krell
- Martin Schulz, LLNL
- William Hachfeld and Dave Whitney, Krell
- Dane Gardner, Argo Navis/Krell
- Chris Chambreau and Matt Legendre, LLNL
- Dyninst group (Bart Miller, UW & Jeff Hollingsworth, UMD)
- Phil Roth, ORNL
- Ciera Jaspan, CMU



❖ Welcome

- 1) Concepts in performance analysis**
- 2) Introduction into Open | SpeedShop**
- 3) How to gather and understand profiles?**
- 4) How to relate data to architectural properties?**
- 5) How to understand I/O tradeoffs?**
- 6) How to detect bottlenecks in Parallel Codes**
- 7) Comparing performance results.**
- 8) User Interfaces (GUI, CLI, python, scripting)**
- 9) Static binary support**
- 10) How can I repeat this at home?**
- 11) Other projects related to Open | SpeedShop**

❖ Conclusions

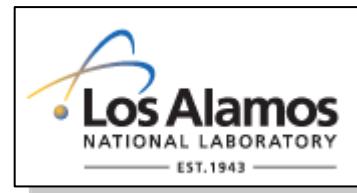
Open | SpeedShop™

Section 1 Concepts in Performance Analysis

SciDAC 2011 Tutorial

How to Analyze the Performance of Parallel Codes 101

A case study with Open|SpeedShop



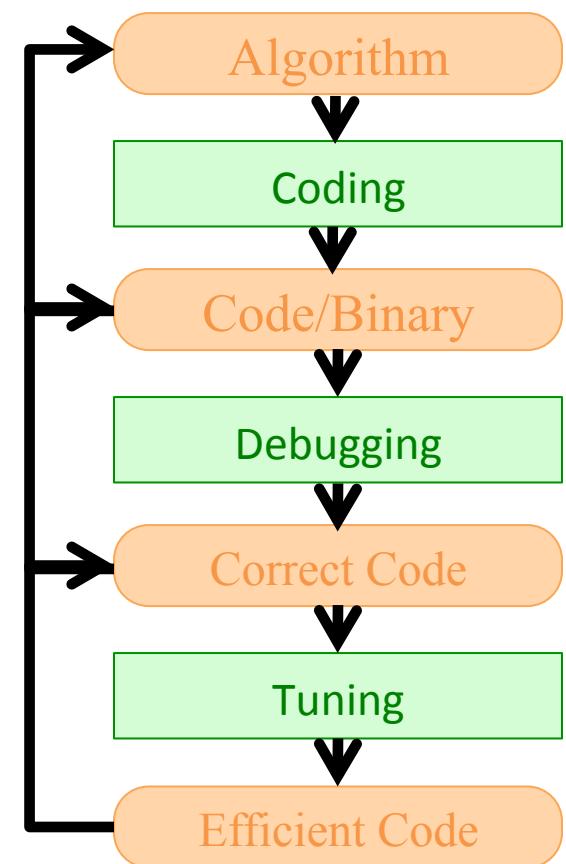
Typical Development Cycle

❖ Performance tuning is an essential part of the development cycle

- Potential impact at every stage
 - Message patterns
 - Data structure layout
 - Algorithms
- Should be done from early on in the life of a new HPC code

❖ Typical use

- Measure performance
- Analyze data
- Modify code and/or algorithm
- Repeat measurements
- Analyze differences



❖ First line of defense

- Full execution timings (UNIX: “time” command)
- Comparisons between input parameters
- Keep and track historical trends

❖ Disadvantages

- Measurements are coarse grain
- Can’t pin performance bottlenecks

❖ Alternative: code integration of performance probes

- Hard to maintain
- Requirements significant a priori knowledge

❖ Performance Tools

- Enable fine grain instrumentation
- Show relation to source code
- Work universally across applications

What to Look For: Sequential Runs

❖ Step 1: Identify computational intensive parts

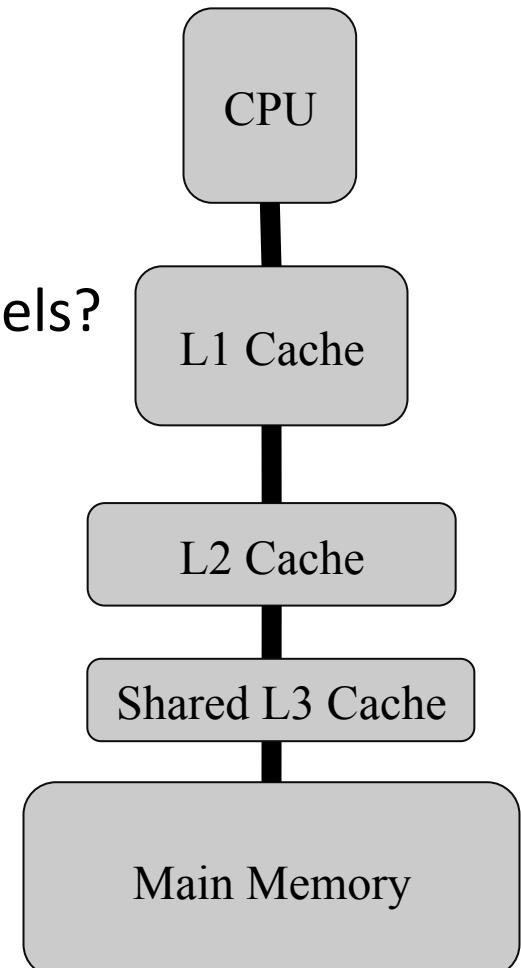
- Where am I spending my time?
 - Modules/Libraries
 - Statements
 - Functions
- Is the time spent in the computational kernels?
- Does this match my intuition?

❖ Impact of memory hierarchy

- Do I have excessive cache misses?
- How is my data locality?
- Impact of TLB misses?

❖ External resources

- Is my I/O efficient?
- Time spent in system libraries?



What to Look For: Shared Memory

❖ Shared memory model

- Single shared storage
- Accessible from any CPU

❖ Common programming models

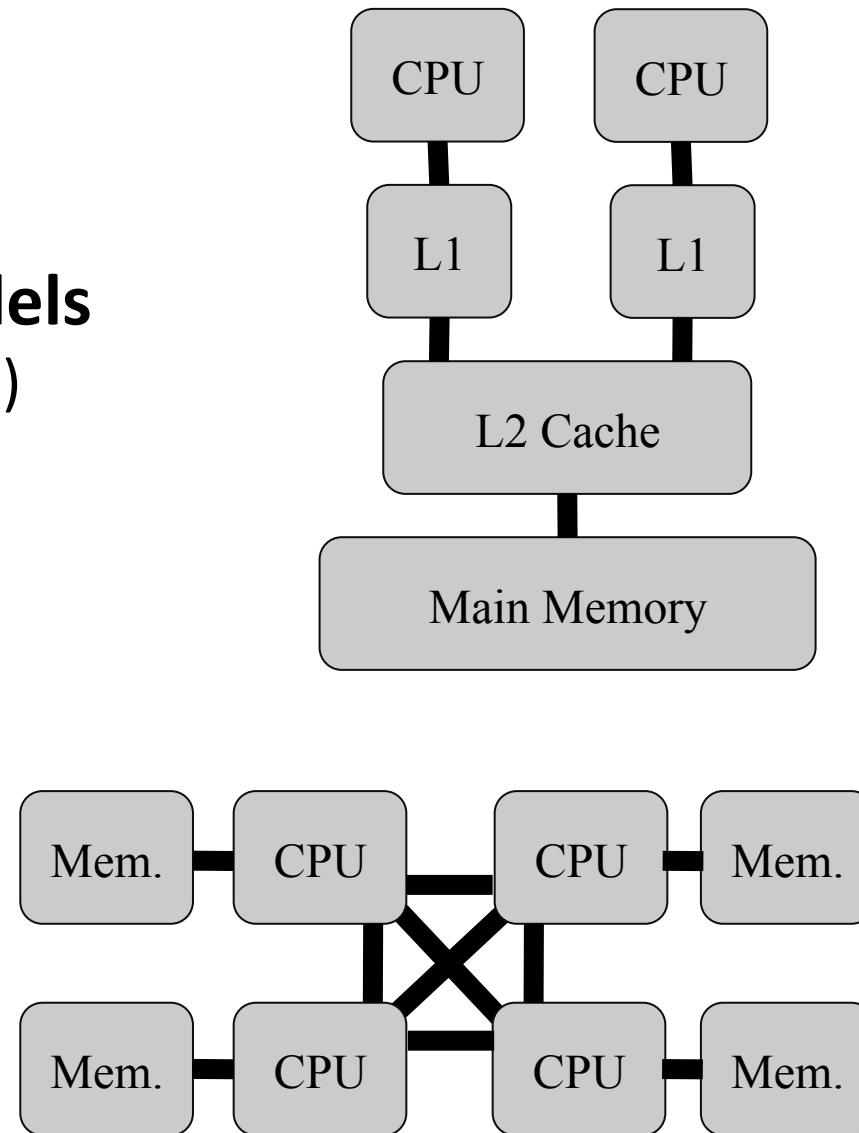
- Explicit threads (e.g., Pthreads)
- OpenMP

❖ Typical performance issues

- Limited bus bandwidth
- Synchronization overhead
- Thread startup
- Limited work per thread

❖ Complications: NUMA

- Memory locality critical
- Thread:Memory assignments



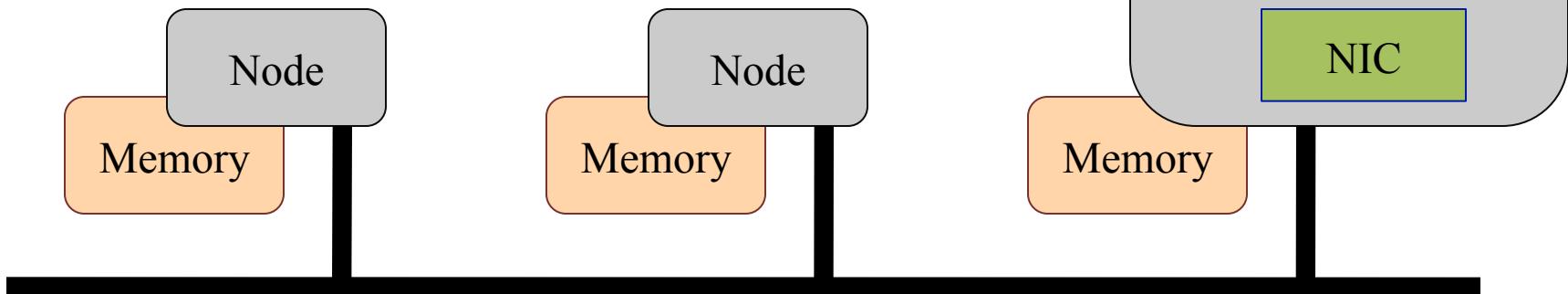
What to Look For: Message Passing

❖ Distributed Memory Model

- Sequential/shared memory nodes coupled by a network
- Only local memory access
- Data exchange using message passing (e.g., MPI)

❖ Typical performance issues

- Long blocking times waiting for data
- Low message rates
- Limited network bandwidth
- Global collective operations



❖ Statistical Sampling

- Periodically interrupt execution and record location
- Report statistical distribution across all reported locations
- Data typically aggregated (all results added together) over time
- Most common metric is time, but other metrics possible
- Useful to get an overview
- Low and uniform overhead

❖ Event Tracing

- Gather and store individual application events
- Examples: function invocations, MPI messages, I/O calls
- Events are typically time stamped
- Provides detailed per event information
- Can lead to huge data volumes
- Higher overhead, potentially in bursts

How to Select a Tool?

❖ A tools must have the right features

- Which question should be answered?
- How deep do I want to analyze the code?

❖ A tool must match the application's workflow

- Requirements from instrumentation technique
 - Access to and knowledge about source code? Recompilation time?
 - Machine environments? Supported platforms?
- Remote execution/analysis?

❖ Local support support and installation

❖ Why We Picked Open|SpeedShop?

- Sampling and tracing in a single framework
- Easy to use GUI & command line options for remote execution
- Transparent instrumentation (preloading & binary)
 - No need to recompile application
- Extensible

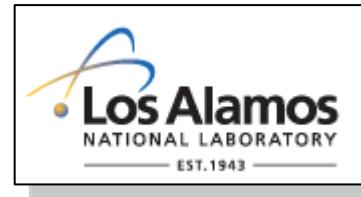
Open | SpeedShop™

Section 2 Introduction into Open|SpeedShop

SciDAC 2011 Tutorial

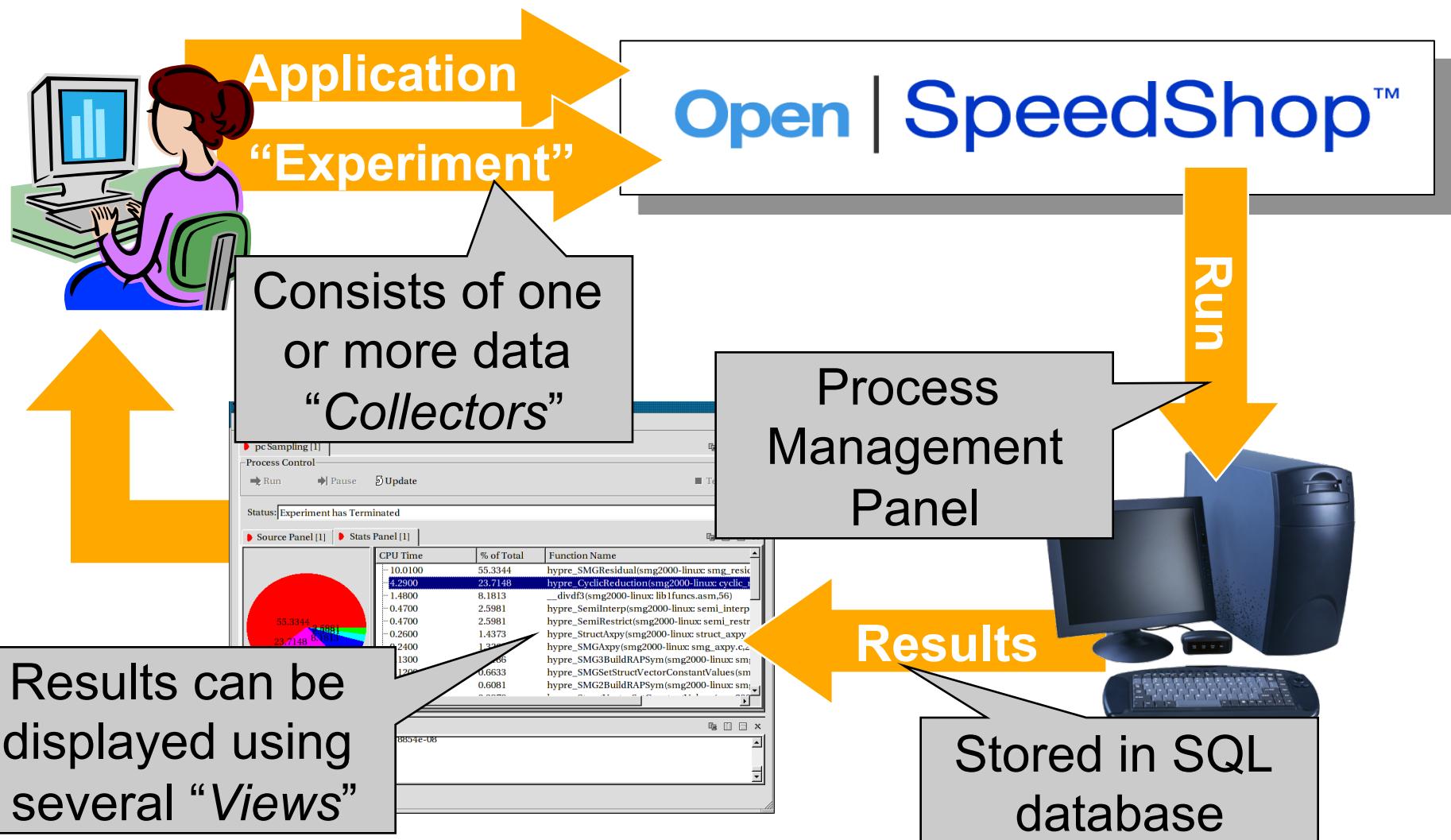
How to Analyze the Performance of Parallel Codes 101

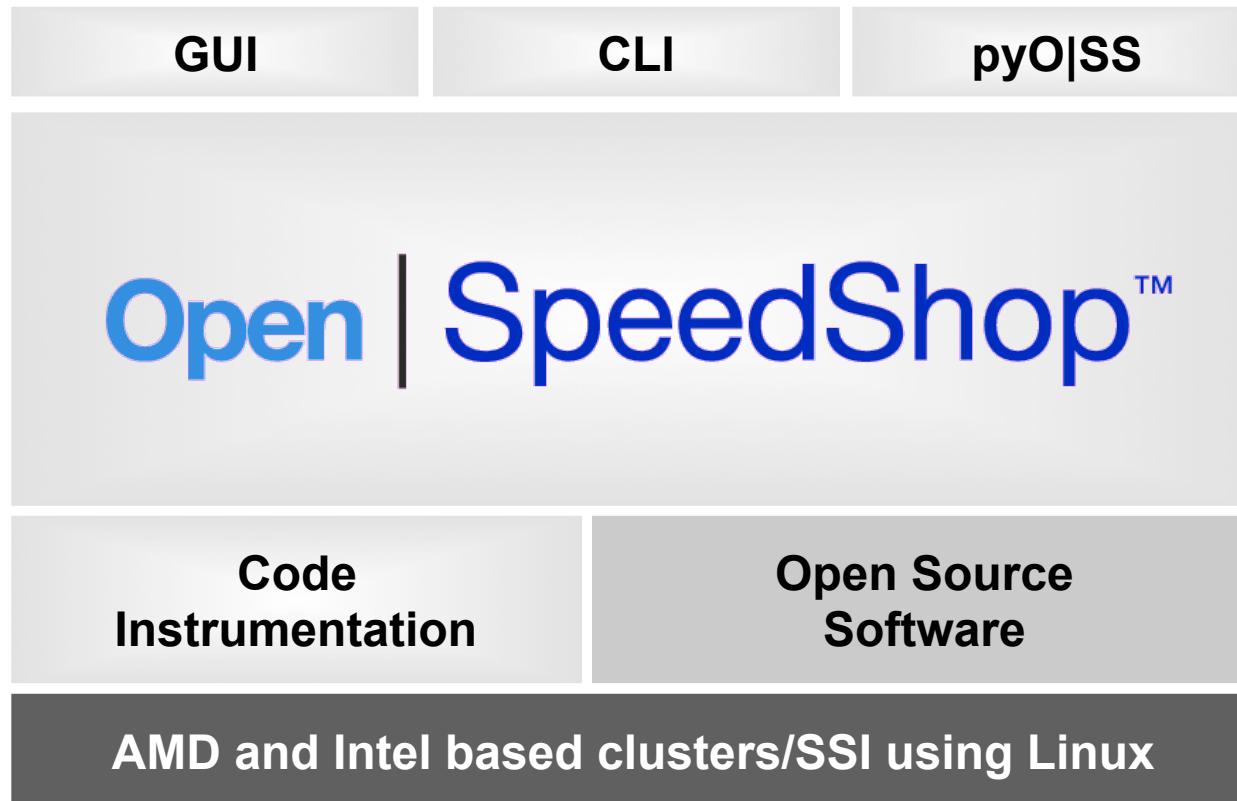
A case study with Open|SpeedShop



Experiment Workflow

Open|SpeedShop Workflow



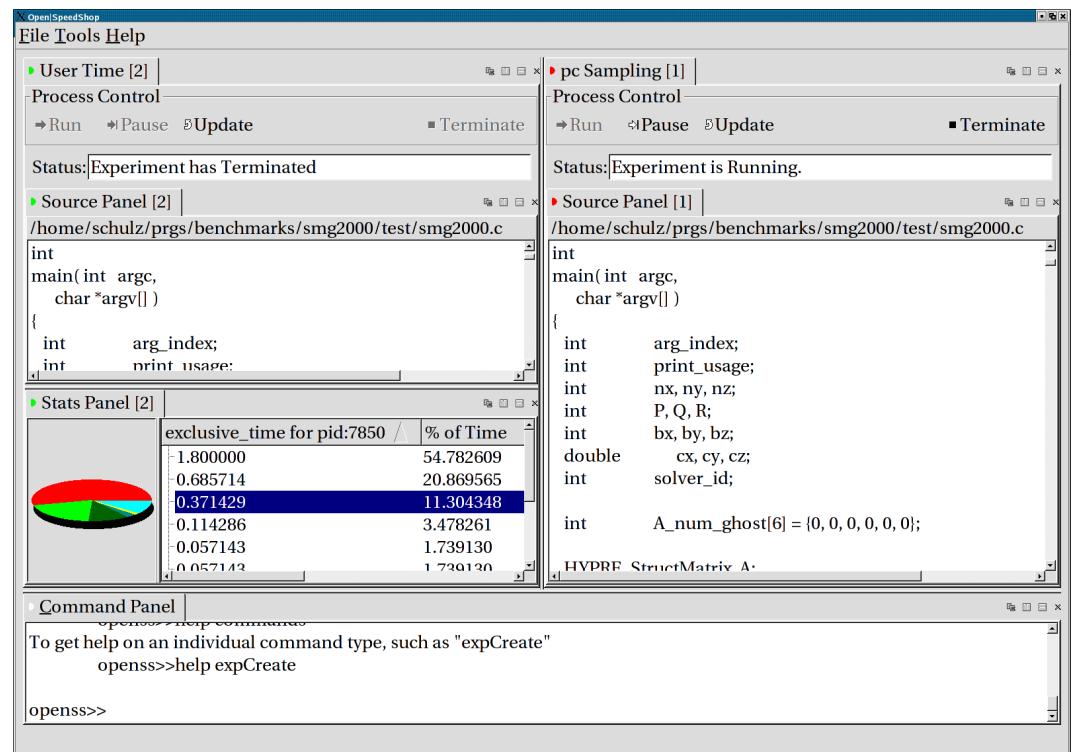


❖ Step 1

- Gather data from command line
- Example: osspcsamp “<application>”
- Create database

❖ Step 2

- Analyze data in GUI
- Simple graphics
- Relate data to source
- openss –f <db file>



❖ Scripting language

- Immediate command interface
- O|SS interactive command line (CLI)

Experiment Commands

expAttach
expCreate
expDetach
expGo
expView

List Commands

list -v exp

```
import openss

my_filename=openss.FileList("myprog.a.out")
my_exptype=openss.ExpTypeList("pcsamp")
my_id=openss.expCreate(my_filename,my_exptype)

openss.expGo()

My_metric_list = openss.MetricList("exclusive")
my_viewtype = openss.ViewTypeList("pcsamp")
result = openss.expView(my_id,my_viewtype,my_metric_list)
```

❖ Concept of an Experiment

- What to measure and what to analyze?
- Experiment type (data type to gather) is chosen by user
- Any experiment can be applied to any application

❖ Consists of Collectors and Views

- Collectors define specific performance data sources
 - Program counter samples
 - Call stack samples
 - Hardware counters
 - Tracing of library routines
- Views specify performance data aggregation and presentation
- Multiple collectors per experiment possible

❖ PC Sampling (**pcsample**)

- Record PC in user defined time intervals
- Low overhead overview of time distribution
- Good first step, lightweight overview

❖ Call Path Profiling (**usertime**)

- PC Sampling and Call stacks for each sample
- Provides inclusive and exclusive timing data
- Use to find hot call paths, whom is calling who

❖ Hardware Counters (**hwc**, **hwctime**, **hwcsamp**)

- Access to data like cache and TLB misses
- hwc, hwctime:
 - Sample a HWC event based on an event threshold
 - Default event is PAPI_TOT_CYC overflows
- hwcsamp:
 - Sample up to six events based on a sample time (hwcsamp)
 - Default events are PAPI_FP_OPS and PAPI_TOT_CYC

❖ Input/Output Tracing (io, iot)

- Record invocation of all POSIX I/O events
- Provides aggregate and individual timings
- Provide argument information for each call (iot)

❖ MPI Tracing (mpi, mpit, mpiotf)

- Record invocation of all MPI routines
- Provides aggregate and individual timings
- Provide argument information for each call (mpit)
- Create Open Trace Format (OTF) output (mpiotf)

❖ Floating Point Exception Tracing (fpe)

- Triggered by any FPE caused by the application
- Helps pinpoint numerical problem areas

❖ O|SS supports MPI and threaded codes

- Tested with a variety of MPI implementations
- Thread support based on POSIX threads
- OpenMP supported through POSIX threads

❖ Any experiment can be applied to parallel application

- Automatically applied to all tasks/threads
- Default views aggregate across all tasks/threads
- Data from individual tasks/threads available

❖ Specific parallel experiments (e.g., MPI)

- Wraps MPI calls and reports
 - MPI routine time
 - MPI routine parameter information

1. Picking the experiment

- What do I want to measure?
- We will start with pcsamp to get a first overview

2. Launching the application

- How do I control my application under O|SS?
- Enclose how you normally run your application in quotes
- **osspcsamp “mpirun –np 256 smg2000 –n 65 65 65”**

3. Storing the results

- O|SS will create a database
- Name: smg2000-pcsamp.openss

4. Exploring the gathered data

- O|SS will print a default report
- Open the GUI to analyze data in detail (run: “**openss**”)

Example Run with Output

❖ osspcsample "mpirun -np 2 smg2000 -n 65 65 65"

```
osspcsamp "mpirun -np 2 ./smg2000 -n 65 65 65"
[openss]: pcsamp experiment using the pcsamp experiment default sampling rate: "100".
[openss]: Using OPENSS_PREFIX installed in /opt/OSS-mrnet
[openss]: Setting up offline raw data directory in /tmp/jeg/offline-oss
[openss]: Running offline pcsamp experiment using the command:
"mpirun -np 2 /opt/OSS-mrnet/bin/ossrun "./smg2000 -n 65 65 65" pcsamp"
```

Running with these driver parameters:

```
(nx, ny, nz) = (65, 65, 65)
(Px, Py, Pz) = (2, 1, 1)
(bx, by, bz) = (1, 1, 1)
(cx, cy, cz) = (1.000000, 1.000000, 1.000000)
(n_pre, n_post) = (1, 1)
dim      = 3
solver ID = 0
```

Struct Interface:

Struct Interface:

```
wall clock time = 0.049847 seconds
cpu clock time = 0.050000 seconds
```

Example Run with Output

❖ osspcsample “mpirun –np 2 smg2000 –n 65 65 65”

```
=====
```

Setup phase times:

```
=====
```

SMG Setup:

wall clock time = 0.635208 seconds

cpu clock time = 0.630000 seconds

```
=====
```

Solve phase times:

```
=====
```

SMG Solve:

wall clock time = 3.987212 seconds

cpu clock time = 3.970000 seconds

Iterations = 7

Final Relative Residual Norm = 1.774415e-07

[openss]: Converting raw data from /tmp/jeg/offline-oss into temp file X.0.openss

Processing raw data for smg2000

Processing processes and threads ...

Processing performance data ...

Processing functions and statements ...

Example Run with Output

❖ osspcsample “mpirun –np 2 smg2000 –n 65 65 65”

[openss]: Restoring and displaying default view for:

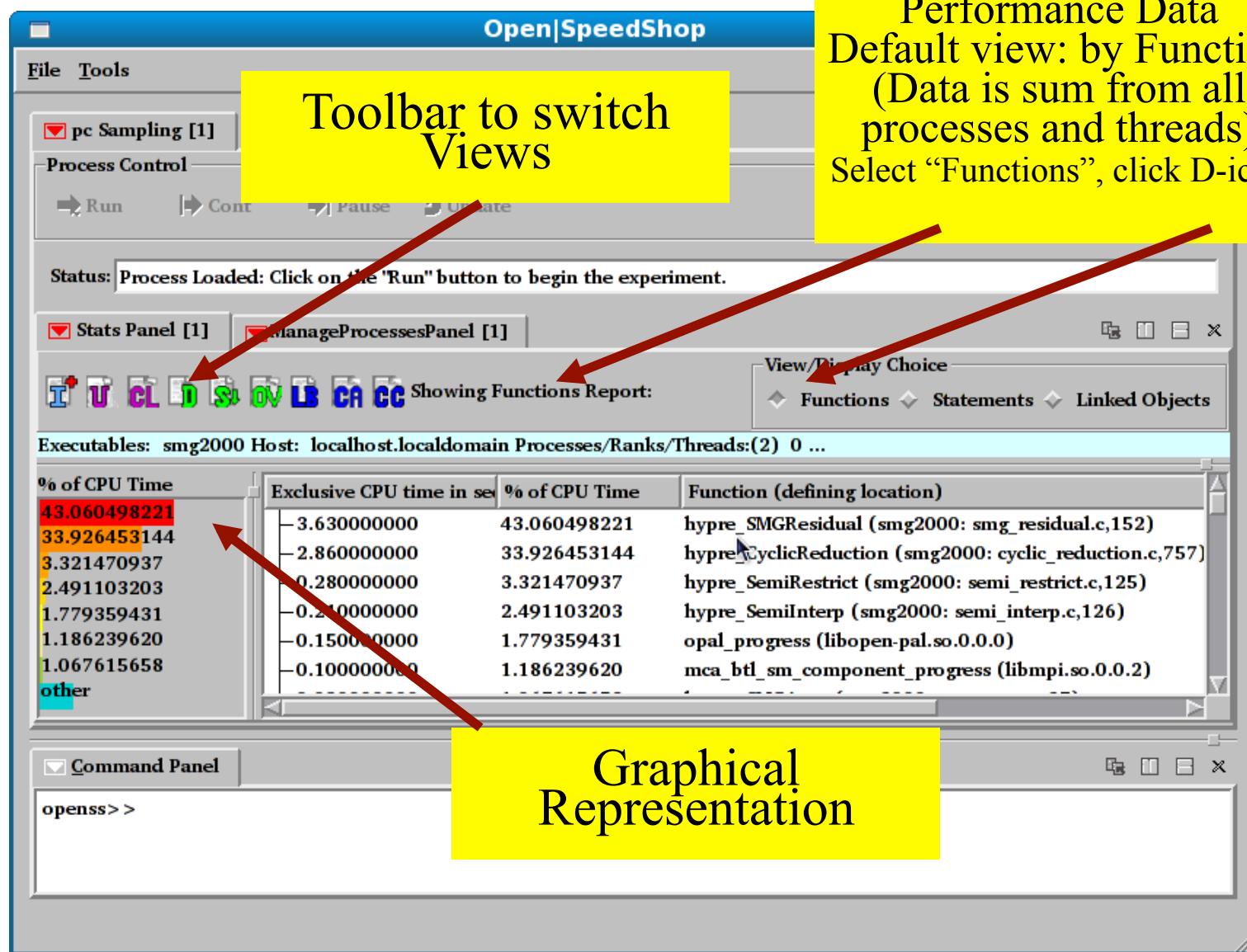
/home/jeg/DEMOS/demos/mpi/openmpi-1.4.2/smg2000/test/smg2000-pcsamp-1.openss

[openss]: The restored experiment identifier is: -x 1

| Exclusive CPU time in seconds. | % of CPU Time | Function (defining location) |
|-----------------------------------|---------------|---|
| 3.630000000 | 43.060498221 | hypre_SMGResidual (smg2000: smg_residual.c,152) |
| 2.860000000 | 33.926453144 | hypre_CyclicReduction (smg2000: cyclic_reduction.c,757) |
| 0.280000000 | 3.321470937 | hypre_SemiRestrict (smg2000: semi_restrict.c,125) |
| 0.210000000 | 2.491103203 | hypre_SemiInterp (smg2000: semi_interp.c,126) |
| 0.150000000 | 1.779359431 | opal_progress (libopen-pal.so.0.0.0) |
| 0.100000000 | 1.186239620 | mca_btl_sm_component_progress (libmpi.so.0.0.2) |
| 0.090000000 | 1.067615658 | hypre_SMGAxpy (smg2000: smg_axpy.c,27) |
| 0.080000000 | 0.948991696 | ompi_generic_simple_pack (libmpi.so.0.0.2) |
| 0.070000000 | 0.830367734 | __GI_memcpy (libc-2.10.2.so) |
| 0.070000000 | 0.830367734 | hypre_StructVectorSetConstantValues (smg2000: struct_vector.c,537) |
| 0.060000000 | 0.711743772 | hypre_SMG3BuildRAPSym (smg2000: smg3_setup_rap.c,233) |

❖ View with GUI: openss –f smg2000-pcsamp-1.openss

Default Output Report View



Open|SpeedShop

File Tools

pc Sampling [1]

Process Control

Run Cont Pause Update

Status: Process Loaded: Click on the 'Run' button to begin the experiment.

Stats Panel [1] ManageProcessesPanel [1]

I U CL D S OV LB CA CC Showing Functions Report:

View/Display Choice

Functions Statements Linked Objects

Executables: smg2000 Host: localhost.localdomain Processes/Ranks/Threads:(2) 0 ...

| % of CPU Time | Exclusive CPU time in sec | % of CPU Time | Function (defining location) |
|---------------|---------------------------|---------------|---|
| 43.060498221 | -3.630000000 | 43.060498221 | hypre_SMGResidual (smg2000: smg_residual.c,152) |
| 33.926453144 | -2.860000000 | 33.926453144 | hypre_CyclicReduction (smg2000: cyclic_reduction.c,757) |
| 3.321470937 | 0.280000000 | 3.321470937 | hypre_SemiRestrict (smg2000: semi_restrict.c,125) |
| 2.491103203 | -0.210000000 | 2.491103203 | hypre_SemiInterp (smg2000: semi_interp.c,126) |
| 1.779359431 | -0.150000000 | 1.779359431 | opal_progress (libopen-pal.so.0.0.0) |
| 1.186239620 | -0.100000000 | 1.186239620 | mca_btl_sm_component_progress (libmpi.so.0.0.2) |
| 1.067615658 | | | |
| other | | | |

Command Panel

openss> >

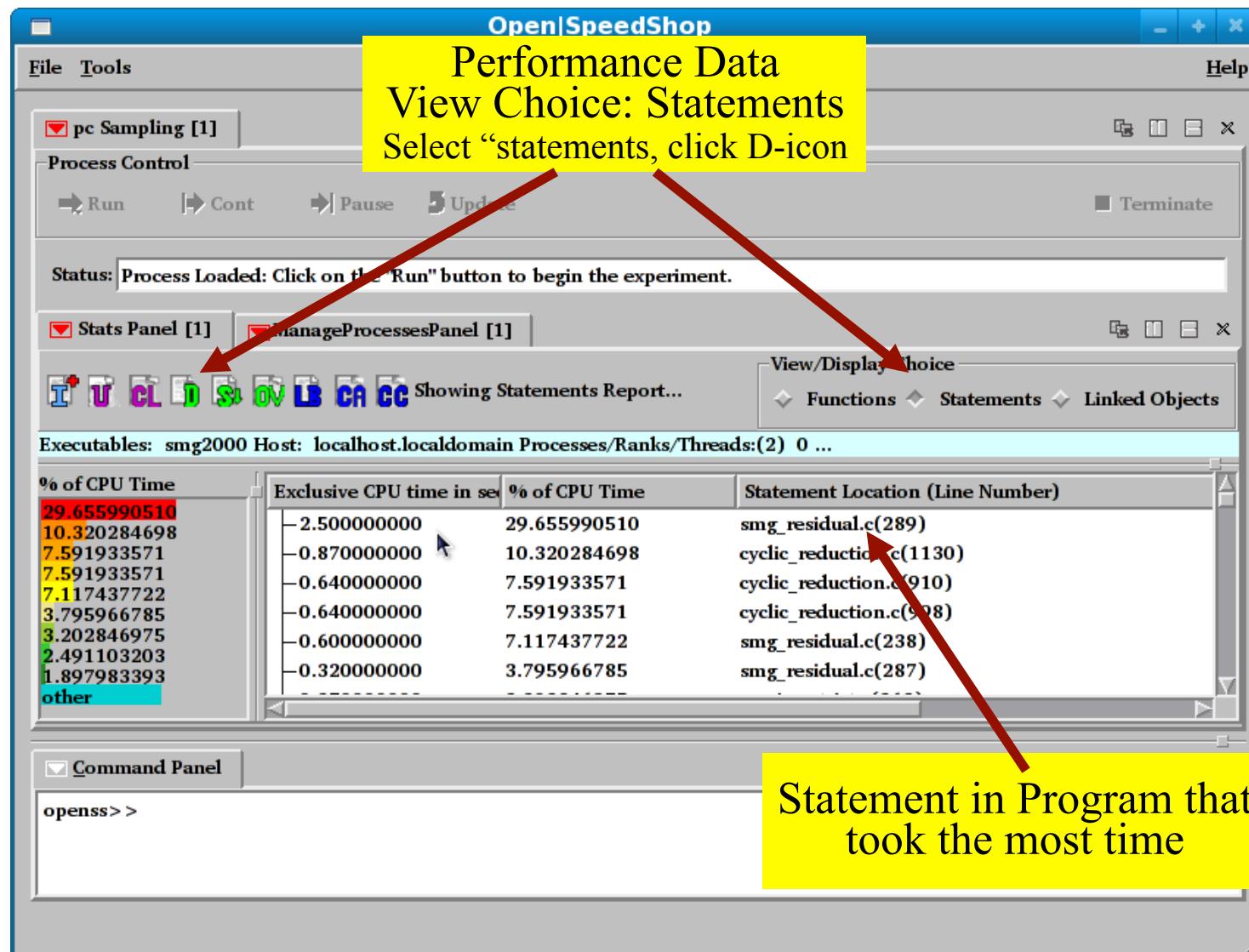
Toolbar to switch Views

Performance Data
Default view: by Function
(Data is sum from all processes and threads)
Select "Functions", click D-icon

Showing Functions Report

Graphical Representation

Statement Report Output View



Associate Source & Performance Data

Double click to open source window

Use window controls to split/arrange windows

Status: Process Loaded: Click on the "Run" button to begin the experiment.

Stats Panel [1] Source Panel [1]

Exclusive CPU time (% of CPU Time) Statement Location (Line)

| Exclusive CPU time | % of CPU Time | Statement Location (Line) |
|--------------------|---------------|---------------------------|
| 2.500000000 | 29.655990510 | smg_residual.c(289) |
| -0.870000000 | 10.320284698 | cyclic_reduction.c(113) |
| -0.640000000 | 7.591933571 | cyclic_reduction.c(910) |
| -0.640000000 | 7.591933571 | cyclic_reduction.c(998) |
| -0.600000000 | 7.117437722 | smg_residual.c(238) |
| -0.320000000 | 3.795966785 | smg_residual.c(287) |
| -0.270000000 | 3.202846975 | semi_restrict.c(262) |
| -0.210000000 | 2.491103203 | topo.Unity_componen |

Command Panel ManageProcessesPanel [1]

Processes: Rank

| Process | Rank |
|---------|------|
| 30947 | 0 |
| 30948 | 1 |

Process Sets: Dynamic Process Set

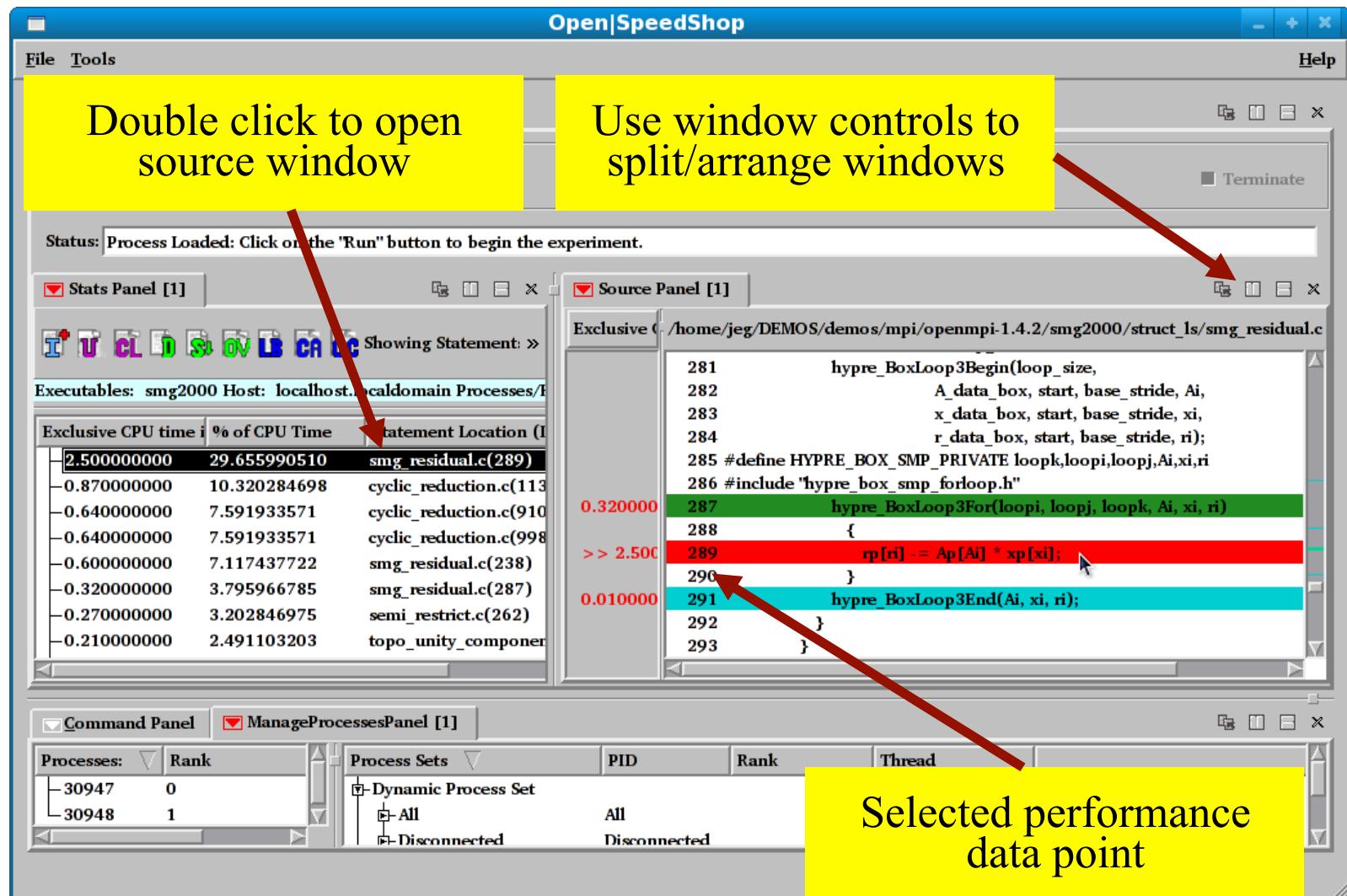
- All
- Disconnected

PID Rank Thread

/home/jeg/DEMONS/demos/mpi/openmpi-1.4.2/sm2000/struct_ls/sm2000.c

```
281     hypre_BoxLoop3Begin(loop_size,
282                         A_data_box, start, base_stride, Ai,
283                         x_data_box, start, base_stride, xi,
284                         r_data_box, start, base_stride, ri);
285 #define HYPRE_BOX_SMP_PRIVATE loopk,loopj,Ai,xi,ri
286 #include 'hypre_box_smp_forloop.h'
287     hypre_BoxLoop3For(loopi, loopj, loopk, Ai, xi, ri)
288 {
289     rp[ri] -= Ap[Ai] * xp[xi];
290 }
291     hypre_BoxLoop3End(Ai, xi, ri);
292 }
293 }
```

Selected performance data point



❖ The take away messages are:

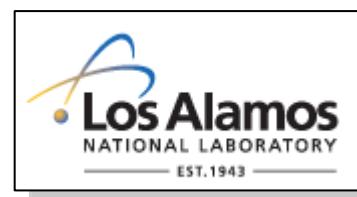
- Place **the way you run your application normally** in quotes and pass it as an argument to osspcsamp, or any of the other experiment convenience scripts: ossio, ossmpi, etc.
 - osspcsamp “srun –N 8 –n 64 ./mpi_application app_args”
- Open|SpeedShop creates default view to stdout
- Open|SpeedShop creates a database file
- Display alternative views of the data with the GUI via:
 - openss –f <database file>
- Display alternative views of the data with the CLI via:
 - openss –cli –f <database file>
- On clusters, need to set OPENSS_RAWDATA_DIR
 - More on this later – usually done in a module or dotkit file.
- Start with pcsamp for overview of performance
- Then home into performance issues with other experiments, if necessary

Open | SpeedShop™

Section 3

How to Gather and Understand Profiles?

SciDAC 2011 Tutorial
How to Analyze the Performance of Parallel Codes
A case study with Open/SpeedShop



❖ What is a profile?

- Aggregate (sum) measurements during collection
- Over time and code sections

❖ Why use a profile?

- Reduced size of performance data
- Typically collected with low overhead
- Provides good overview of performance of application

❖ Disadvantages of using a profile

- Requires a-priori definition of aggregation
- Omits performance details of individual events
- Possible sampling frequency skew

❖ Statistical Performance Analysis

- Interrupt execution in periodic intervals
- Record location of execution (Program Counter value)
- Optionally annotate with additional data
 - Stack traces
 - Hardware counters
- Count equivalent samples

❖ Advantages

- Low Overhead
- Low Perturbation
- Good for:
 - Getting a program overview
 - Finding program hotspots

❖ PC Sampling

- Approximates CPU Time for Line and Function
- No Call Stacks
- Convenience Script: osspcsamp

❖ User Time

- Inclusive vs. Exclusive CPU Time
- Includes Call Stacks
- Convenience Script: ossusertime

❖ HW Counters

- Samples Hardware Counter Overflows (hwc, hwctime)
- Samples Periodically up to six (6) Hardware Counter Events (hwcsamp)
- Convenience Scripts: osshwc, osshwctime, osshwcsamp

❖ Answers a basic question:

- Where does my code spend its time?

❖ Representation

- List of code elements
 - Varying granularity
 - Statements, Functions, Libraries (linked objects)
- Time spent at each function

❖ Flat profiles through sampling

- Alternative to overhead of direct measurements
- Add contributions taken from samples
- Requires sufficient number of samples

Offline pcsamp experiment on smg2000 application

Option 1: Convenience script basic syntax

```
osspcsamp "smg2000 -n 50 50 50"
```

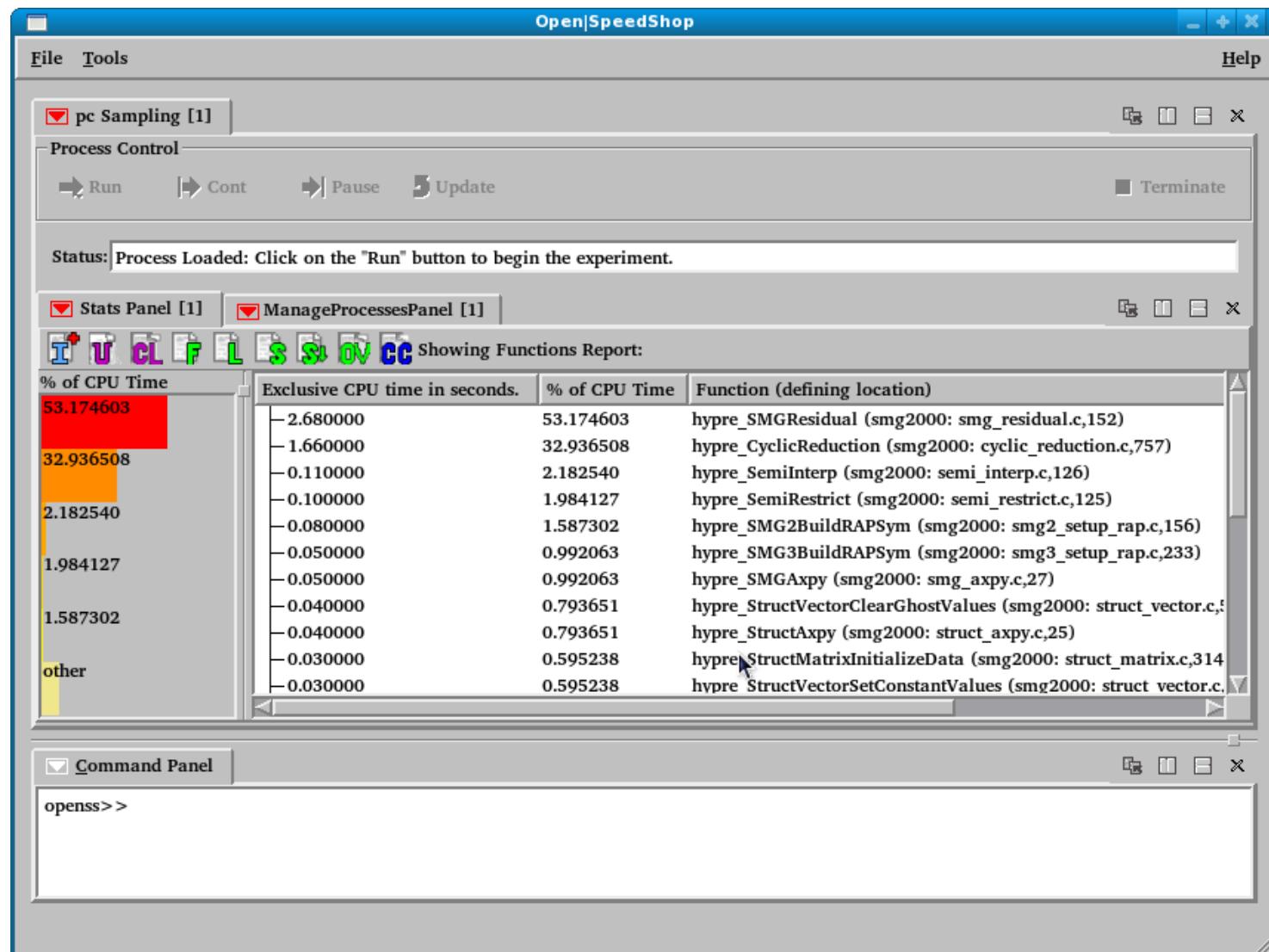
```
osspcsamp "smg2000 -n 50 50 50" high
```

❖ Optional Parameter:

- Sampling frequency (samples per second)
- Alternative parameter: high (200) | low (50) | default (100)

Recommendation: compile code with -g to get statements!

Viewing Flat Profiles



❖ Profiles show computationally intensive code regions

- First views: Time spent per functions or per statements

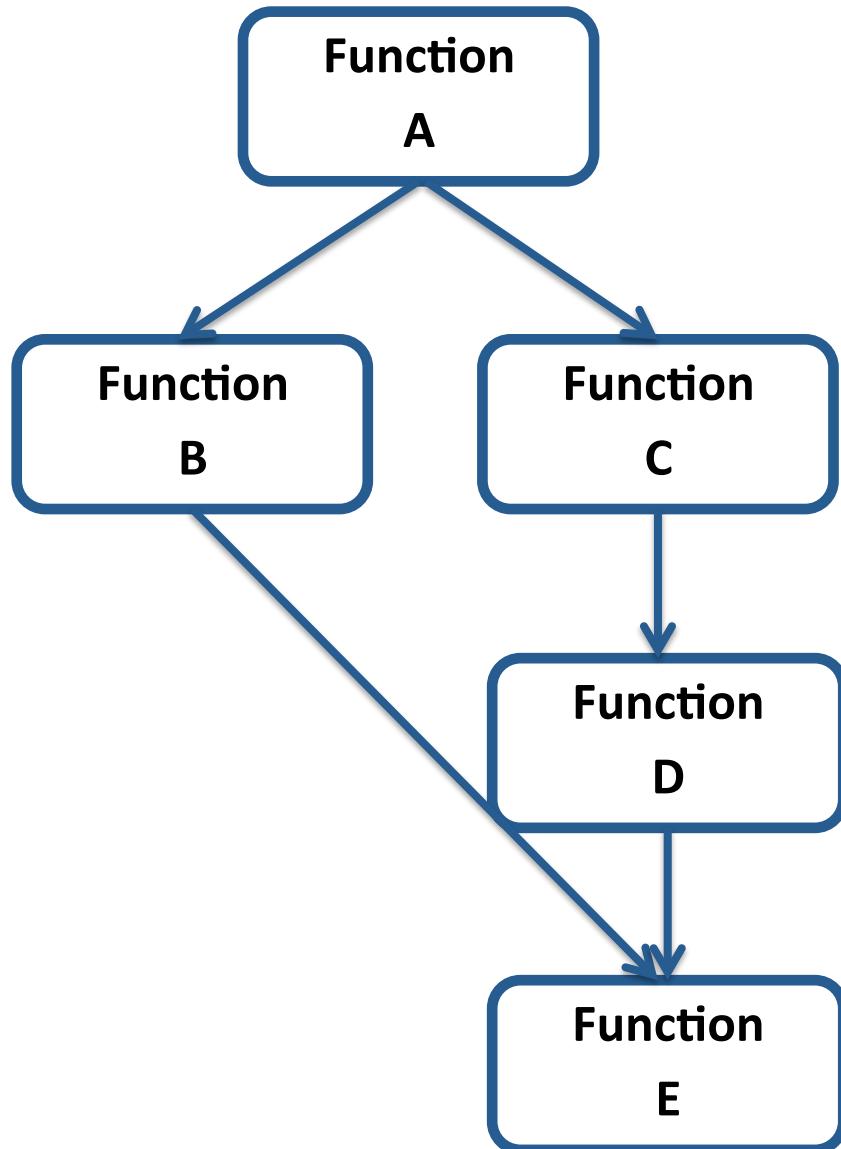
❖ Questions:

- Are those functions/statements expected?
- Do they match the computational kernels?
- Any runtime functions taking a lot of time?

❖ Identify bottleneck components

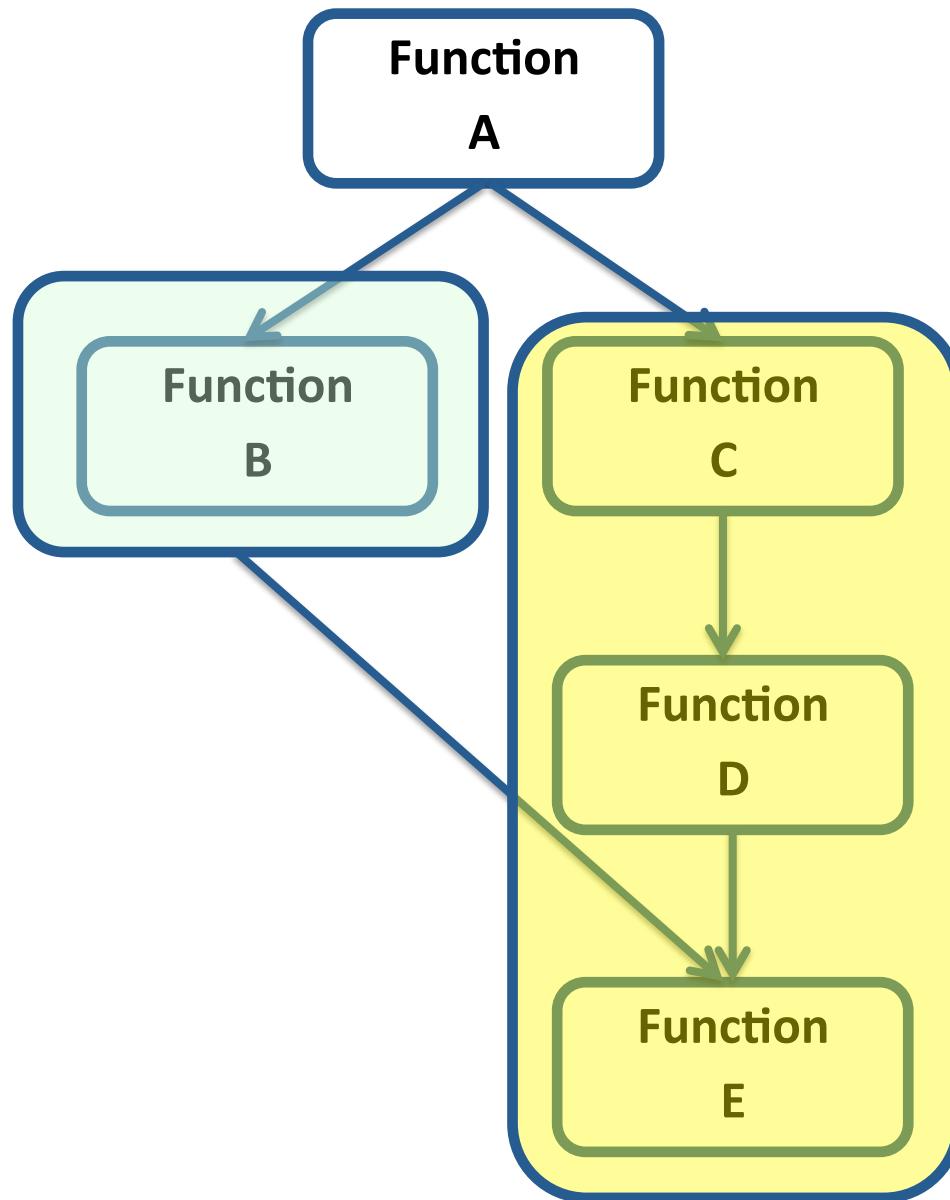
- View the profile aggregated by shared objects
- Correct/expected modules?
- Impact of support and runtime libraries

Adding Context (usertime experiment)



- ❖ **Missing information in flat profiles**
 - Distinguish routines called from multiple callers
 - Understand the call invocation history
 - Context for performance data
- ❖ **Critical technique: Stack traces**
 - Gather stack trace for each performance sample
 - Aggregate only samples with equal trace
- ❖ **User perspective:**
 - Butterfly views (shows caller/callee relationships)
 - Hot call paths
 - Paths through application that take most time

Adding Context (usertime experiment)



- ❖ **Provides inclusive/exclusive time**
 - Time spent inside a function only (exclusive)
 - See: Function B
 - Time spent inside a function and its children (inclusive)
 - See Function C and children
- ❖ **Similar to the pcsamp experiment**
 - Collect program counter information
 - But also: collect call stack information at every sample
- ❖ **Tradeoffs**
 - Obtain additional context information
 - Have higher overhead/lower sampling rate

Offline usertime experiment on smg2000 application

Option 1: Convenience script basic syntax

```
ossusertime "smg2000 -n 50 50 50"
```

```
ossusertime "smg2000 -n 50 50 50" low
```

❖ Parameters

Sampling frequency (samples per second)

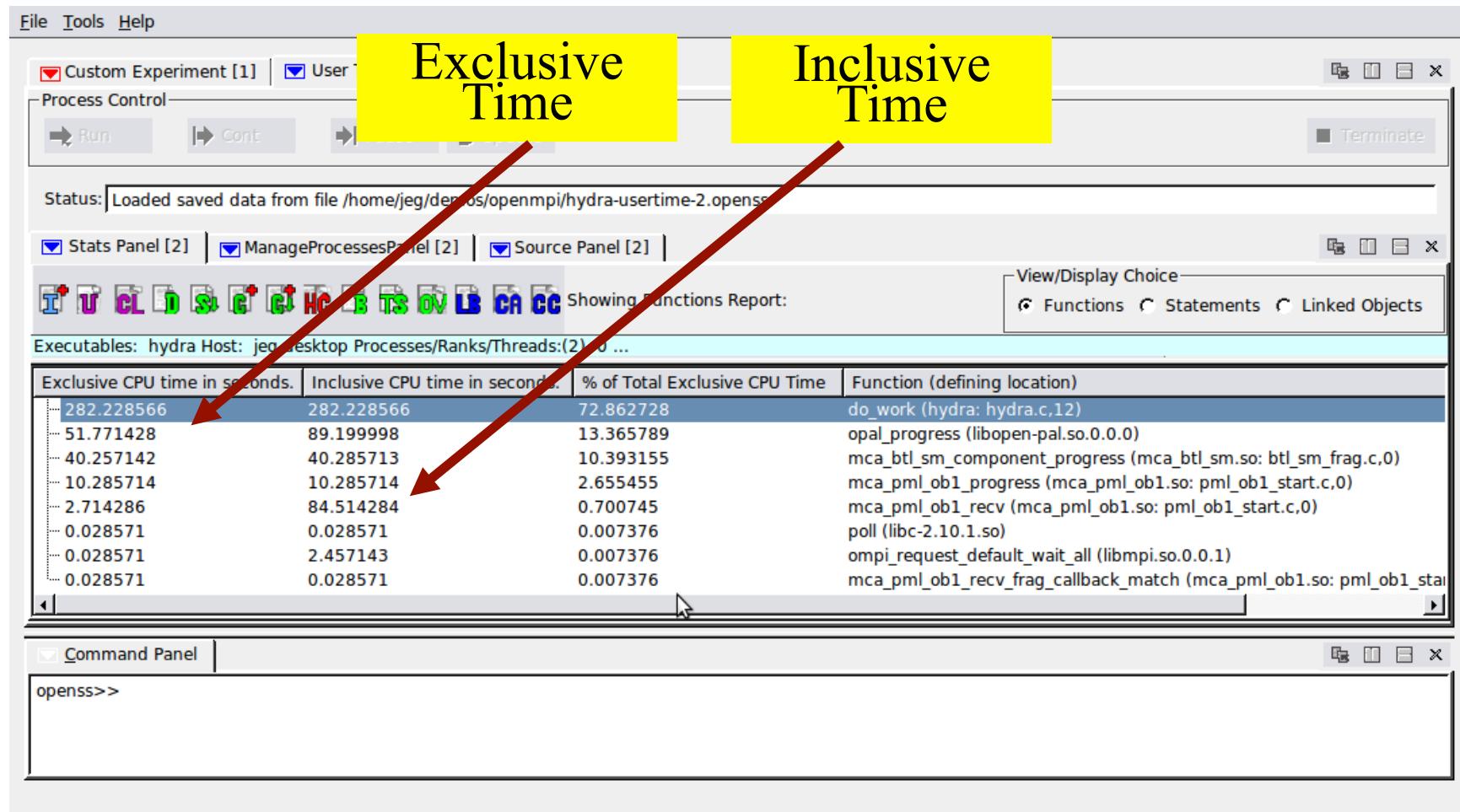
Alternative parameter: high (70) | low (18) | default (35)

Recommendation: compile code with -g to get statements!

Viewing the Usertime Experiment

❖ Default View

- Similar to pcsamp view from first example
- Calculates inclusive versus exclusive times

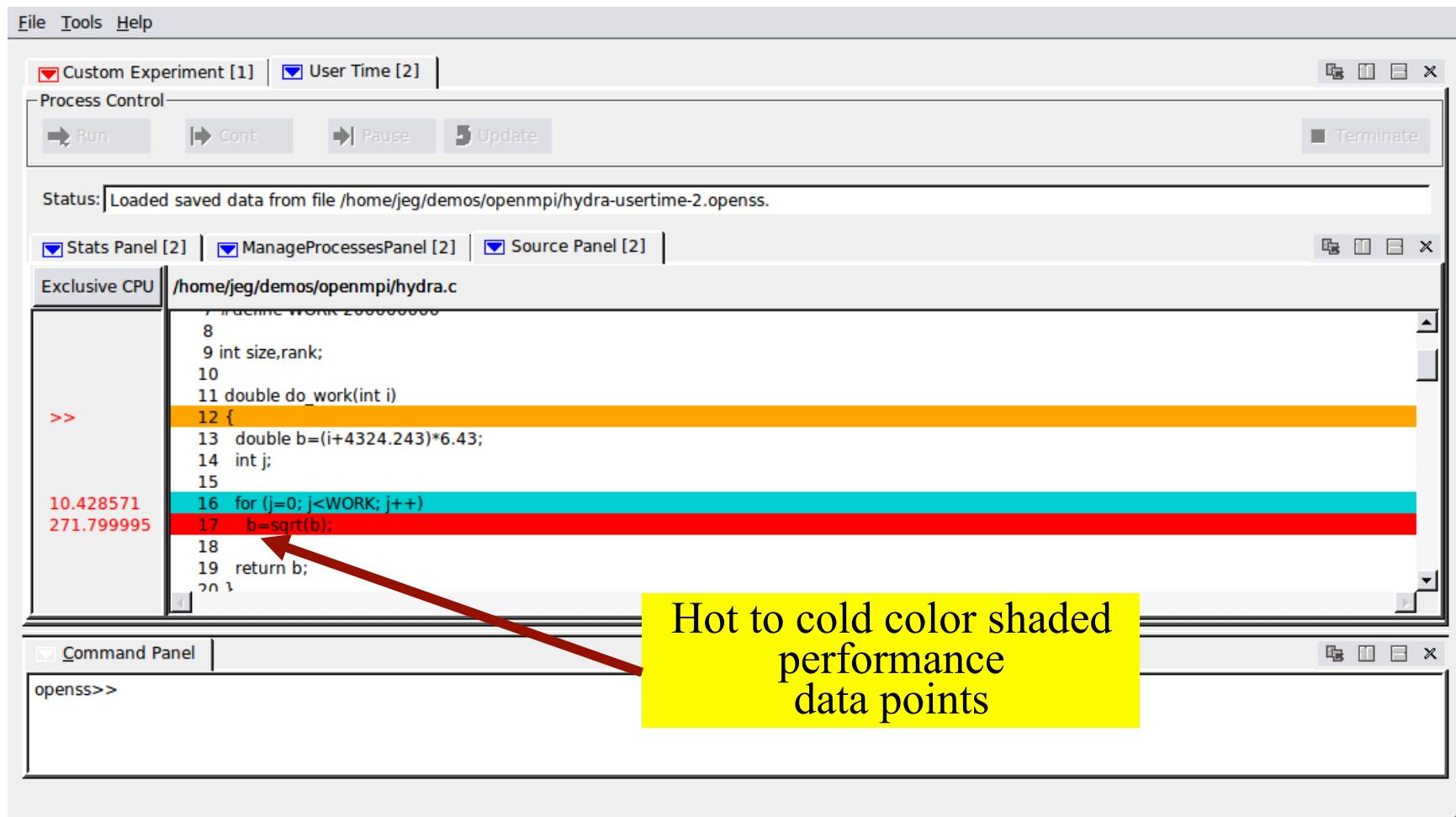


The screenshot shows the SpeedShop interface with the 'Functions' report selected. The 'Exclusive Time' and 'Inclusive Time' columns are highlighted with yellow boxes and red arrows pointing to them.

| Exclusive CPU time in seconds. | Inclusive CPU time in seconds. | % of Total Exclusive CPU Time | Function (defining location) |
|--------------------------------|--------------------------------|-------------------------------|--|
| 282.228566 | 282.228566 | 72.862728 | do_work (hydra: hydra.c,12) |
| 51.771428 | 89.199998 | 13.365789 | opal_progress (libopen-pal.so.0.0.0) |
| 40.257142 | 40.285713 | 10.393155 | mca_btl_sm_component_progress (mca_btl_sm.so: btl_sm_frag.c,0) |
| 10.285714 | 10.285714 | 2.655455 | mca_pml_ob1_progress (mca_pml_ob1.so: pml_ob1_start.c,0) |
| 2.714286 | 84.514284 | 0.700745 | mca_pml_ob1_recv (mca_pml_ob1.so: pml_ob1_start.c,0) |
| 0.028571 | 0.028571 | 0.007376 | poll (libc-2.10.1.so) |
| 0.028571 | 2.457143 | 0.007376 | ompi_request_default_wait_all (libmpi.so.0.0.1) |
| 0.028571 | 0.028571 | 0.007376 | mca_pml_ob1_recv_frag_callback_match (mca_pml_ob1.so: pml_ob1_start.c,0) |

Source Code Mapping

❖ Exclusive CPU time in left column



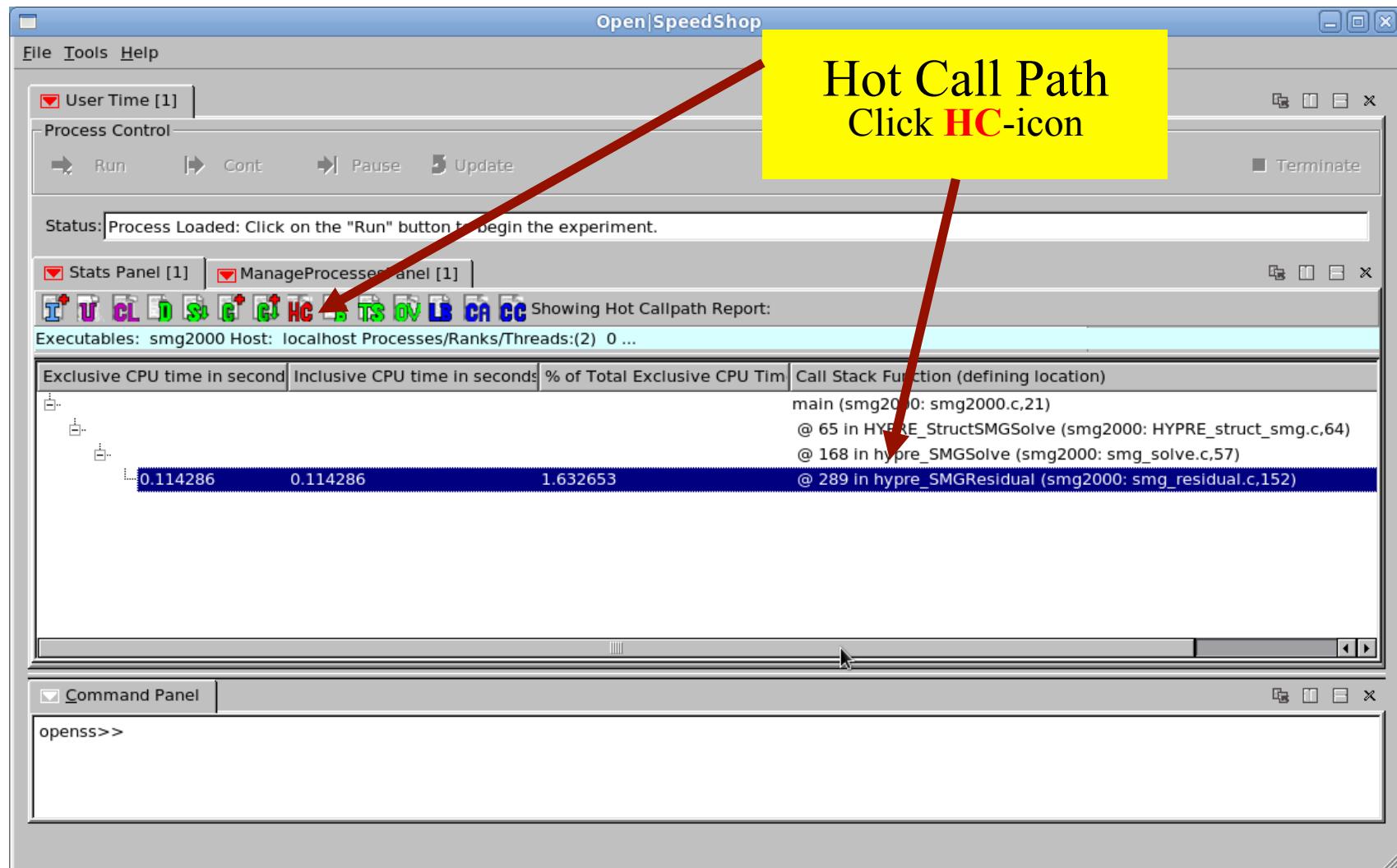
❖ Inclusive versus exclusive times

- If similar: child executions are insignificant
 - May not be useful to profile below this layer
- If inclusive time significantly greater than exclusive time:
 - Focus attention to the execution times of the children

❖ Butterfly analysis

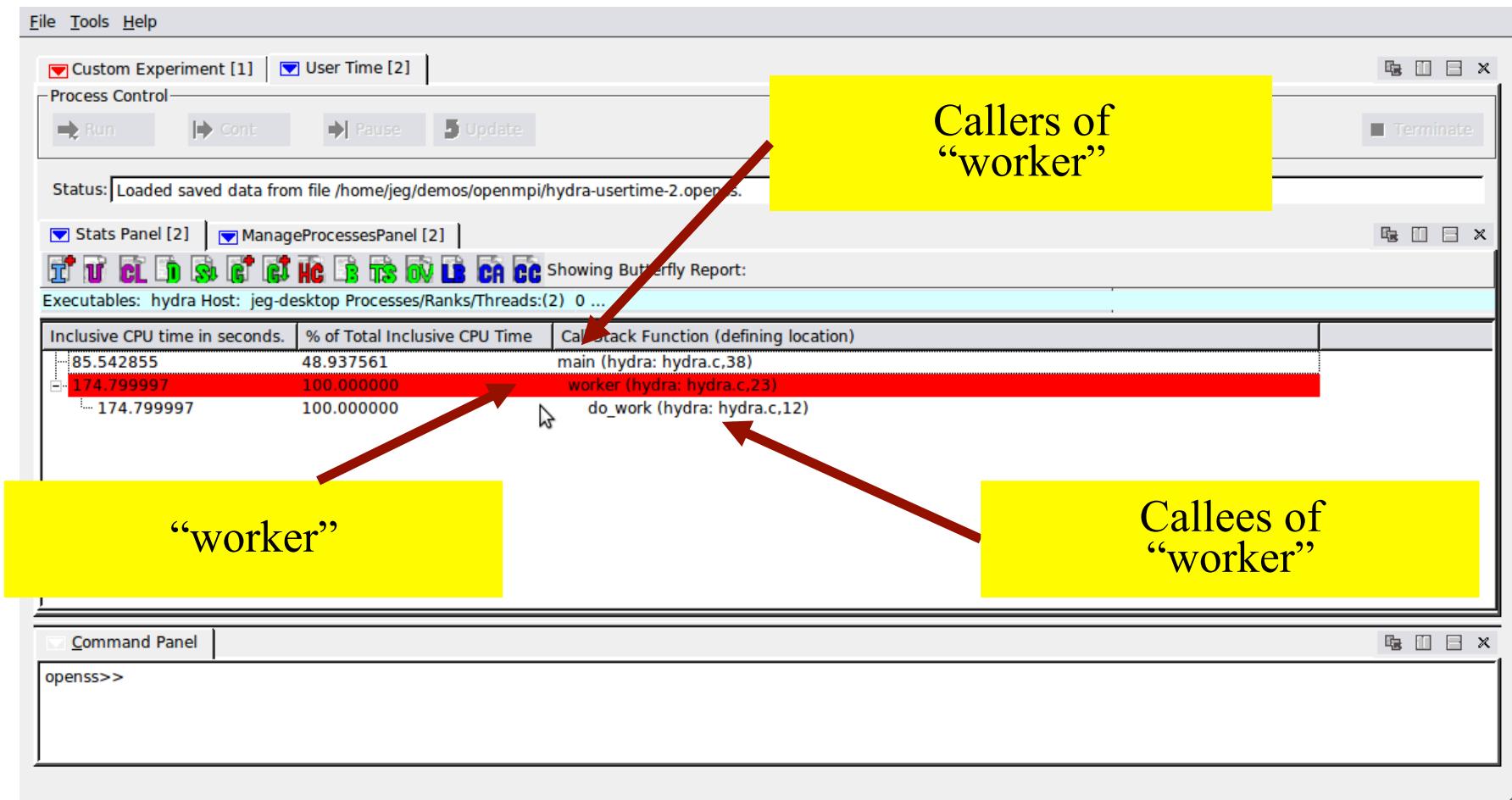
- Should be done on “suspicious” functions
- Shows split of time in callees and callers

Stack Trace Views: Hot Call Path



Stack Trace Views: Butterfly View

- ❖ Similar to well known “gprof” tool



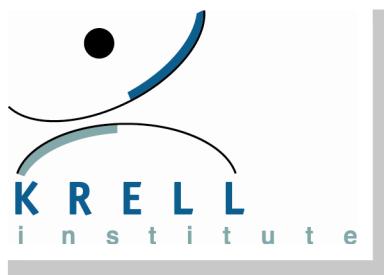
Open | SpeedShop™

Section 4

How to Relate Data to Architectural Properties?

SciDAC 2011 Tutorial

How to Analyze the Performance of Parallel Codes 101
A case study with Open|SpeedShop



❖ **Timing information shows where you spend your time**

- Hot functions / statements /libraries
- Hot call paths

❖ **BUT: It doesn't show you why**

- Are the computationally intensive parts efficient?
- Which resources constrain execution?

❖ **Answer can be very platform dependent**

- Bottlenecks may differ
- Cause of missing performance portability
- Need to tune to architectural parameters

❖ **Next: Investigate hardware/application interaction**

- Efficient use of hardware resources
- Architectural units (on/off chip) that are stressed

❖ Modern memory systems are complex

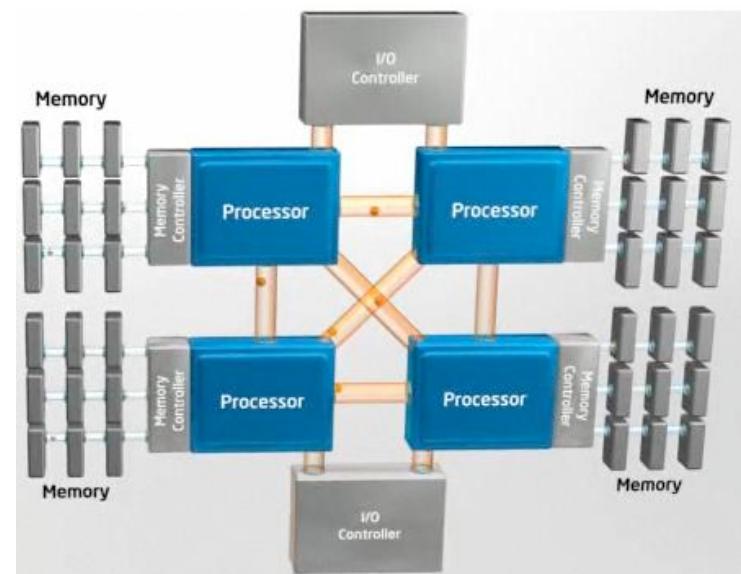
- Deep hierarchies
- Explicitly managed memory
- NUMA behavior
- Streaming/Prefetching

❖ Key: locality

- Accessing the same data repeatedly
- Accessing neighboring data

❖ Information to look for

- Read/Write intensity
- Prefetch efficiency
- Cache miss rate at all levels
- TLB miss rates
- NUMA overheads



❖ Computational intensity

- Cycles per instruction (CPI)
- Number of floating point instructions

❖ Branches

- Number of branches taken (pipeline flushes)
- Miss speculations / Wrong branch prediction results

❖ SIMD/Multimedia/Streaming Extensions

- Are they used in the respective code pieces?

❖ System-wide information

- I/O busses
- Network counters
- Power/Temperature sensors
- Difficulty: relating this information to source code

❖ Architectural Features

- Typically/Mostly packaged inside the CPU
- Count hardware events transparently without overhead

❖ Newer platforms also provide system counters

- Network cards and switches
- Environmental sensors

❖ Drawbacks

- Availability differs between platform & processors
- Slight semantic differences between platforms
- In some cases : requires privileged access & kernel patches

❖ Recommended: Access through PAPI

- API for tools + simple runtime tools
- Abstractions for system specific layers
- <http://icl.cs.utk.edu/papi/>

❖ Provides access to hardware counters

- Implemented on top of PAPI
- Access to PAPI and native counters
- Examples: cache misses, TLB misses, bus accesses

❖ Basic model 1: Thresholding

- User selects counter
- Run until a fixed number of events have been reached
- Take PC sample at that location
- Reset number of events
- Ideal number of events (threshold) depends on application

❖ Basic model 2: Timer Based Sampling

❖ Three versions

- HWC = flat hardware counter profile
- HWCTime = profiles with context / stacktraces
- HWCSamp = PC sampling with multiple HWCs

❖ For `osshwc`, `osshwctime`: Open|SpeedShop supports ...

- Non derived PAPI presets
 - All non derived events reported by “`papi_avail -a`”
 - Also reported by running “`osshwc`” or “`osshwctime`” with no args
- All native events
 - Architecture specific (incl. naming)
 - Names listed in the PAPI documentation

❖ Threshold depends on application

- Overhead vs. Accuracy
 - Higher threshold cause less samples
- Rare events need smaller thresholds (or they are lost)
- Frequent events need large thresholds (or overhead dominates)
- Takes experience and/or trial and error
- Running “`papiex`” can help to get a first idea

Example: Offline hwc[time] experiment on SMG2000

Convenience Script Syntax:

```
osshwc[time] "smg2000" <counter> <threshold>
```

NOTE: If the output is empty, try lowering the <threshold> value. It is 10000 by default. Try 5000. There may not have been enough PAPI event occurrences to record and present.

Available counters:

- Any counter reported by “papi_avail –a” and not derived
- Or, use osshw or osshwctime with no args to see available counters.
- Native counters listed in the PAPI documentation

Examples of Typical Counters

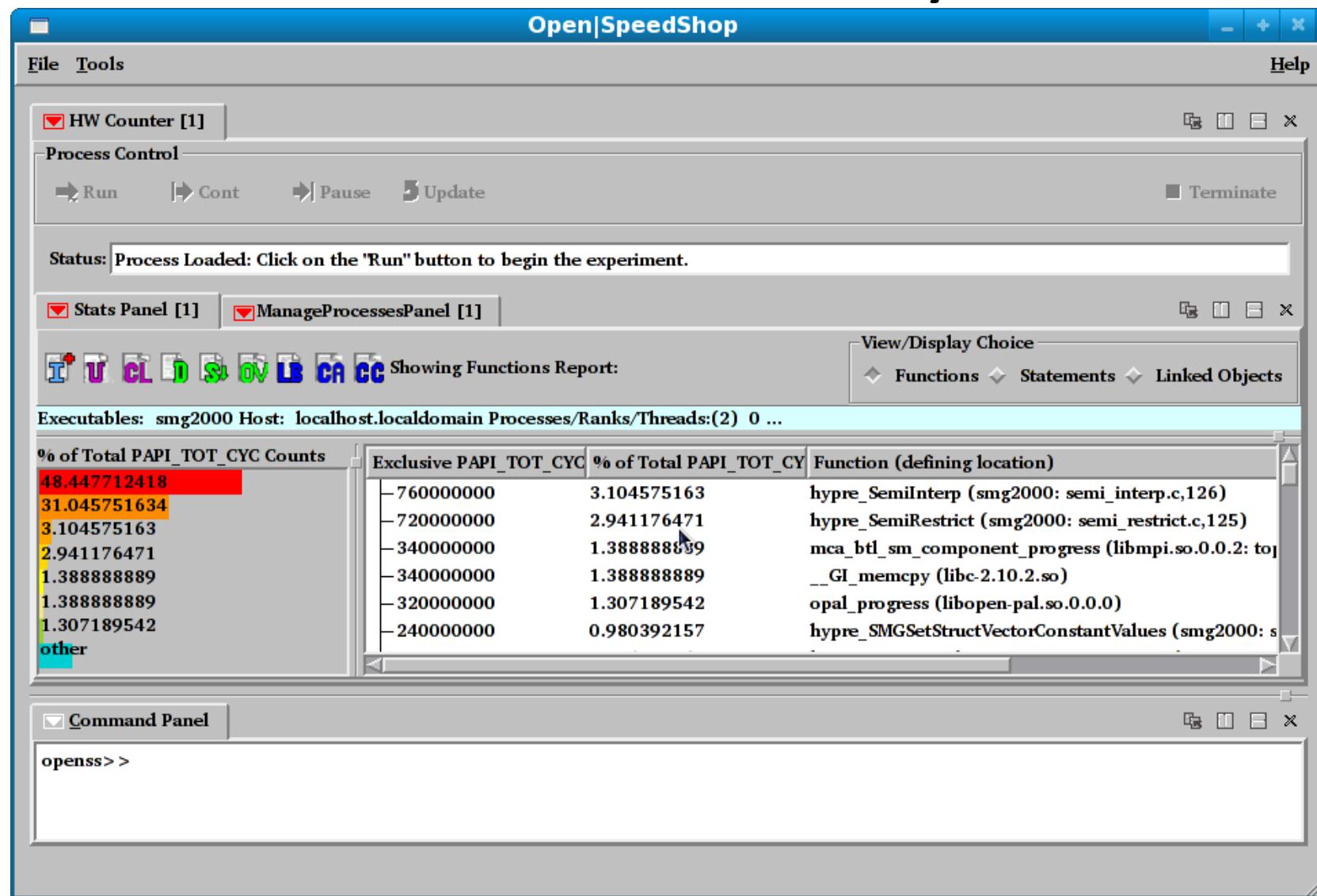
| PAPI Name | Description | Threshold |
|--------------|---------------------------------------|-------------|
| PAPI_L1_DCM | L1 data cache misses | high |
| PAPI_L2_DCM | L2 data cache misses | high/medium |
| PAPI_L1_DCA | L1 data cache accesses | high |
| PAPI_FPU_IDL | Cycles in which FPUs are idle | high/medium |
| PAPI_STL_ICY | Cycles with no instruction issue | high/medium |
| PAPI_BR_MSP | Miss-predicted branches | medium/low |
| PAPI_FP_INS | Number of floating point instructions | high |
| PAPI_LD_INS | Number of load instructions | high |
| PAPI_VEC_INS | Number of vector/SIMD instructions | high/medium |
| PAPI_HW_INT | Number of hardware interrupts | low |
| PAPI_TLB_TL | Number of TLB misses | low |

Note: Threshold indications are just rough guidance and depend on the application.

Note: Not all counters exist on all platforms (check with `papi_avail`)

Viewing hwc Data

❖ hwc default view: Counter = Total Cycles



The screenshot shows the OpenSpeedShop interface with the title "Open|SpeedShop". The main window displays a "Functions Report" with a table of PAPI_TOT_CYC counts. The table has columns for Exclusive PAPI_TOT_CYC, % of Total PAPI_TOT_CYC, and Function (defining location). The top row of the table is highlighted in red.

| % of Total PAPI_TOT_CYC Counts | Exclusive PAPI_TOT_CYC | % of Total PAPI_TOT_CYC | Function (defining location) |
|--------------------------------|------------------------|-------------------------|---|
| 48.447712418 | -760000000 | 3.104575163 | hypre_SemiInterp (smg2000: semi_interp.c,126) |
| 31.045751634 | -720000000 | 2.941176471 | hypre_SemiRestrict (smg2000: semi_restrict.c,125) |
| 3.104575163 | -340000000 | 1.388888889 | mca_btl_sm_component_progress (libmpi.so.0.0.2: topology.c,107) |
| 2.941176471 | -340000000 | 1.388888889 | _GI_memcpy (libc-2.10.2.so) |
| 1.388888889 | -320000000 | 1.307189542 | opal_progress (libopen-pal.so.0.0.0) |
| 1.388888889 | -240000000 | 0.980392157 | hypre_SMGSetStructVectorConstantValues (smg2000: smg.h,126) |
| 1.307189542 | | | |
| other | | | |

The left sidebar shows "Process Control" buttons: Run, Cont, Pause, Update, and Terminate. The bottom command panel shows the text "openss> >".

Viewing hwctime Data

hwctime default view: Counter = L1 Cache Misses

Open|SpeedShop

File Tools Help

HWCTime Panel [1]

Process Control

Run Cont Pause Update Terminate

Status: Process Loaded: Click on the 'Run' button to begin the experiment.

Stats Panel [1] ManageProcessesPanel [1]

View/Display Choice

I U CL D S G C H C B TS OV LB CA CC Showing Statements Report...

Functions Statements Linked Objects

Executables: smg2000 Host: localhost.localdomain Processes/Ranks/Threads:(2) 0 ...

% of Total Exclusive PAPI_L1_DCM Counts

| Exclusive PAPI_L1_DCM | Inclusive PAPI_L1_DCM | % of Total Exclusive PAPI_L1_DCM | Statement Location (Line Number) |
|-----------------------|-----------------------|----------------------------------|----------------------------------|
| 36.729857820 | 232500000 | 36.729857820 | smg_residual.c(289) |
| 8.886255924 | 56250000 | 8.886255924 | cyclic_reduction.c(1130) |
| 8.056872038 | 51000000 | 8.056872038 | smg_residual.c(287) |
| 7.227488152 | 45750000 | 7.227488152 | cyclic_reduction.c(998) |
| 7.109004739 | 45000000 | 7.109004739 | cyclic_reduction.c(910) |
| 3.672985782 | 23250000 | 3.672985782 | smg_residual.c(238) |
| 3.317535545 | 21000000 | 3.317535545 | cyclic_reduction.c(753) |
| 2.843601896 | | | |
| 2.369668246 | | | |
| other | | | |

Command Panel

openss> >

❖ For `osshwcsamp`, Open|SpeedShop supports ...

- Derived and Non derived PAPI presets
 - All derived and non derived events reported by “`papi_avail`”
 - Also reported by running “`osshwcsamp`” with no arguments
 - Ability to sample up to six (6) counters at one time
- All native events
 - Architecture specific (incl. naming)
 - Names listed in the PAPI documentation
 - Native events reported by “`papi_native_avail`”

❖ Sampling rate depends on application

- Overhead vs. Accuracy
 - Lower sampling rate cause less samples

Example: Offline hwcsamp experiment on SMG2000

Convenience Script Basic Syntax

```
osshwcsamp "smg2000" <event list> <sampling rate>
```

NOTE: If a counter does not appear in the output, there may be a conflict in the hardware counters. To find conflicts use:

```
papi_event_chooser PRESET <list of events>
```

Find which counters are available:

- Any counters reported by “papi_avail”
- Or, run osshwcsamp without any arguments to see available counters.
- Native counters listed in the PAPI documentation
 - Native counters also reported by “papi_native_avail”

Possible HWC Combinations To Use*

Xeons

- ❖ **PAPI_FP_INS,PAPI_LD_INS,PAPI_SR_INS** (**load store info, memory bandwidth needs**)
- ❖ **PAPI_L1_DCM,PAPI_L1_TCA** (**L1 cache hit/miss ratios**)
- ❖ **PAPI_L2_DCM,PAPI_L2_TCA** (**L2 cache hit/miss ratios**)
- ❖ **LAST_LEVEL_CACHE_MISSES, LAST_LEVEL_CACHE_REFERENCES** (**L3 cache info**)
- ❖ **MEM_UNCORE_RETIRED:REMOTE_DRAM, MEM_UNCORE_RETIRED:LOCAL_DRAM** (**local/nonlocal memory access**)

For Opterons only

- ❖ **PAPI_FAD_INS,PAPI FML_INS** (**Floating point add mult**)
- ❖ **PAPI_FDV_INS,PAPI FSQ_INS** (**sqrt and divisions**)
- ❖ **PAPI_FP_OPS,PAPI_VEC_INS** (**floating point and vector instructions**)
- ❖ **READ_REQUEST_TO_L3_CACHE:ALL_CORES,L3_CACHE_MISSES:ALL_CORES** (**L3 cache**)

*Credit: Koushik Ghosh, LLNL

How to determine valid combinations?

- ❖ How to determine which PAPI event combinations will work
- ❖ In general combinations that are valid will have to pass a test

papi_event_chooser PRESET event1 event2 .. Eventn

The output for a valid/acceptable combination will contain this:

| | |
|-----------------|--------|
| event_chooser.c | PASSED |
|-----------------|--------|

For example:

papi_event_chooser PRESET PAPI_FP_INS PAPI_LD_INS PAPI_SR_INS

PAPI Version : 4.1.2.1

Vendor string and code : GenuineIntel (1)

Model string and code : Intel Nehalem (21)

CPU Revision : 5.000000

...

...

PAPI_VEC_SP 0x80000069 No Single precision vector/SIMD instructions

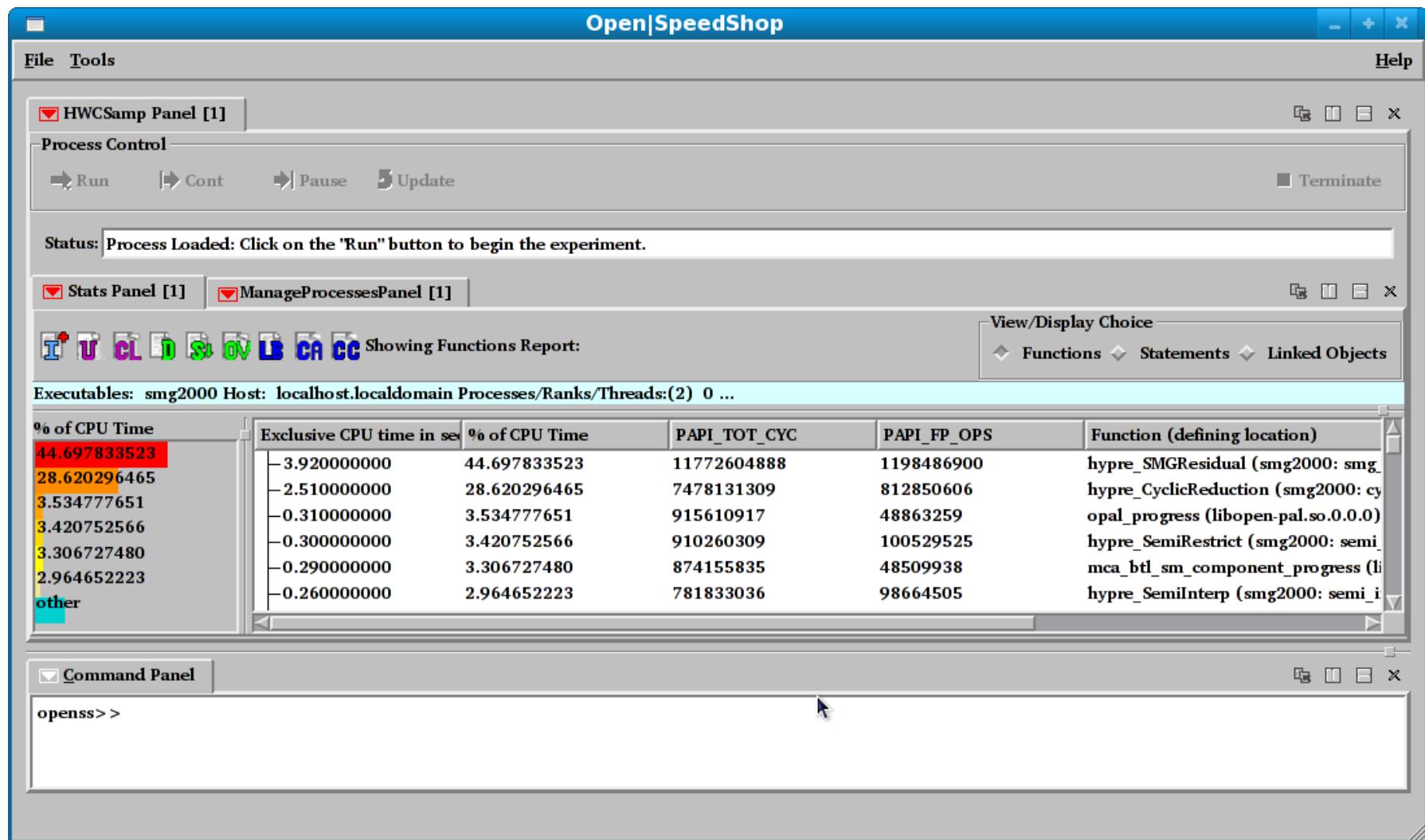
PAPI_VEC_DP 0x8000006a No Double precision vector/SIMD instructions

Total events reported: 44

| | |
|-----------------|--------|
| event_chooser.c | PASSED |
|-----------------|--------|

Viewing hwcsamp Data

❖ hwcsamp default view: Counter = Total Cycles & FP ops



Viewing hwcsamp data in CLI

openss -cli -f smg2000-hwcsamp-1.openss

openss>>[openss]: The restored experiment identifier is: -x 1

openss>>**expview**

| Exclusive CPU time in seconds. | % of CPU Time | PAPI_TOT_CYC | PAPI_FP_OPS Function (defining location) |
|-----------------------------------|---------------|--------------|---|
| 3.920000000 | 44.697833523 | 11772604888 | 1198486900 hypre_SMGResidual (smg2000: smg_residual.c,152) |
| 2.510000000 757) | 28.620296465 | 7478131309 | 812850606 hypre_CyclicReduction (smg2000: cyclic_reduction.c, |
| 0.310000000 | 3.534777651 | 915610917 | 48863259 opal_progress (libopen-pal.so.0.0.0) |
| 0.300000000 | 3.420752566 | 910260309 | 100529525 hypre_SemiRestrict (smg2000: semi_restrict.c,125) |
| 0.290000000 | 3.306727480 | 874155835 | 48509938 mca_btl_sm_component_progress (libmpi.so.0.0.2) |

openss>>**expview -m flops**

| Mflops Function (defining location) |
|--|
| 478.639000000 hypre_ExchangeLocalData (smg2000: communication.c,708) |
| 456.405900000 hypre_StructApxy (smg2000: struct_axpy.c,25) |
| 452.758600000 mca_pml_ob1_isend (libmpi.so.0.0.2) |
| 447.734300000 MPI_Irecv (libmpi.so.0.0.2) |

Viewing hwcsamp data in CLI

openss>>expview -m flops -v statements

Mflops Statement Location (Line Number)

478.639000000 communication.c(734)

462.420300000 smg3_setup_rap.c(677)

456.405900000 struct_axpy.c(69)

453.119050000 smg3_setup_rap.c(672)

450.492600000 semi_restrict.c(246)

444.427800000 struct_matrix.c(29)

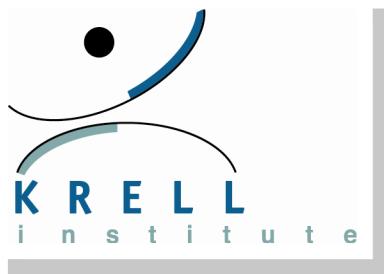
openss>>expview -v linkedobjects

| Exclusive CPU time | % of CPU Time | PAPI_TOT_CYC | PAPI_FP_OPS | LinkedObject |
|--------------------|---------------|--------------|-------------|----------------------|
| in seconds. | | | | |
| 7.710000000 | 87.315968290 | 22748513124 | 2396367480 | smg2000 |
| 0.610000000 | 6.908267271 | 1789631493 | 126423208 | libmpi.so.0.0.2 |
| 0.310000000 | 3.510758777 | 915610917 | 48863259 | libopen-pal.so.0.0.0 |
| 0.200000000 | 2.265005663 | 521249939 | 46127342 | libc-2.10.2.so |
| 8.830000000 | 100.000000000 | 25975005473 | 2617781289 | Report Summary |

openss>>

Open | SpeedShop™

Hardware Performance Counters and their use



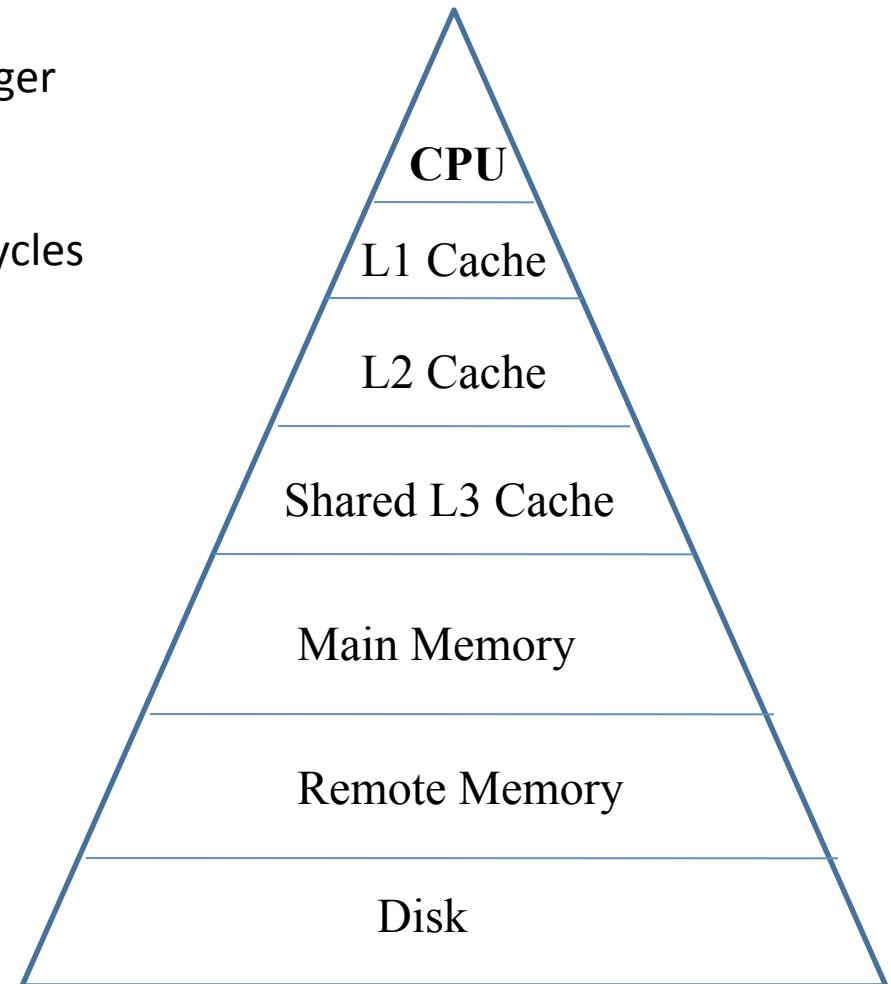
Mahesh Rajan



- 1) Where do the flops go?
- 2) Bring out importance of simple Hardware Counter Metrics (HCM) through easy to understand BALS1, BLAS2, BLAS3 examples
- 3) Use simple Matrix-Matrix multiply example to illustrate HPC node performance optimization (TBD)

The Memory Pyramid and Performance

- Memory closer to the **CPU** faster, smaller
- Memory further away from CPU slower, larger
- Most expensive operation is moving data
- Can only do useful work on data at the top
- Example Nehalem: access latencies clock cycles
 - L1 = 4
 - L2 = 9
 - L3 = 47
 - Main local NUMA Memory = 81
 - Main non-Local NUMA Memory = 128



For a given algorithm, serial performance is all about maximizing CPU Flop rate and minimizing memory operations in scientific codes

BLAS Operations Illustrate impact of moving data

A, B, C = nxn Matrices; x,y = nx1 Vectors; k = Scaler

| Level | Operation | # Memory Refs or Ops | # Flops | Flops/Ops | Comments on Flops/Ops |
|-------|--------------|----------------------|---------|-----------|------------------------|
| 1 | $y = kx + y$ | 3n | 2n | 2/3 | Achieved in Benchmarks |
| 2 | $y = Ax + y$ | n^2 | $2n^2$ | 2 | Achieved in Benchmarks |
| 3 | $C = AB + C$ | $4n^2$ | $2n^3$ | $n/2$ | Exceeds HW MAX |

Use these Flops/Ops to understand how sections of your code relate to simple memory access patterns as typified by these BLAS operations

Example 1: BLAS Level 1

Experiment Conducted: *hwc* or *hwcsamp* with to get the following PAPI Counters:
PAPI_FP_OPS, PAPI_TOT_CYC, PAPI_LD_INS, PAPI_ST_INS, PAPI_TOT_INS

Derived metrics of Interest: GLOPS, Float_ops/cycle, Instructions/cycle, Loads/Cycle, Stores/Cycle, and Flops/memory Ops

Blas 1 Kernel: DAXPY; $y = \text{alpha} * x + y$

Kernel Code: (n=10,000) looped 1000, 000 times for timing purposes

```
do i=1,n  
    y(i) = alpha * x(i) + y(i)  
enddo
```

Example 1: BLAS 1 PAPI data and analysis

| n | Mem Refs=3n | FLOPS Calc | loop blas code | PAPI_LD_INS | PAPI_SR_INS | PAPI_FP_OPS | PAPI_TOT_CYC | PAPI_TOT_INS |
|-------|-------------|------------|----------------|-------------|-------------|-------------|--------------|--------------|
| 10000 | 30000 | 20000 | 100000 | 1.02E+09 | 5.09E+08 | 1.03E+09 | 2.04E+09 | 2.43E+09 |

| code time, secs | code GFLOPS | FPC | IPC | LPC | SPC | Error PAPI FLOPS | Error Corrected FLOPS | Error Mem Refs | PAPI_GLOPS | PAPI Flops/ops | Calc FLOPS/OPS |
|--------------------|-------------|-------------|-------------|-------------|-------------|---------------------|--------------------------|----------------|-------------|-------------------|-------------------|
| 6.4596E-06 | 3.096124 | 0.505386876 | 1.190989226 | 0.500489716 | 0.249412341 | -93.80% | 3.10% | -2.15% | 3.195244288 | 0.673937178 | 0.6666667 |

Processors have a FMADD instruction. Although this instruction performs two Floating Point operations, it is counted as one Floating Point instruction in PAPI. Because of this, there are situations where PAPI_FP_INS may produce fewer Floating Point counts than expected. In this example PAPI_FP_OPS was multiplied by 2 to match the theoretical expected FLOP count.

Formula for calculating Load Instructions:

= (2 vectors)*(vec_length)*(loop)*(bytes_per_word)*(8 bits_per_byte)/(128 BITS_per_load)

What Hardware Counter Metrics can tell us about code performance

- ❖ **As set of useful metrics that can be calculated for code functions:**
 - FLOPS/Memory Ops (FMO) – Would like this to be large – implies good data locality (Also called Computational Intensity or Ops/Refs)
 - Flops/Cycle (FPC) -- Large values for floating point intensive codes suggests efficient CPU utilization
 - Instructions/Cycle (IPC) – Large values suggest good balance with minimal stalls
 - Loads/Cycle (LPC) – Useful for calculating FMO – may indicate good stride through arrays
 - Stores/Cycle (SPC) – Useful for calculating FMO – may indicate good stride through array

| Operation | Kernel | PAPI_GFLOPS | FMO | FPC | IPC | LPC | SPC |
|------------------------------|-------------|-------------|------|------|------|------|------|
| BLAS 1; $y = \alpha * x + y$ | doloop | 0.67 | 0.67 | 0.51 | 1.19 | 0.50 | 0.25 |
| BLAS 2; $y = A * x + y$ | doloop | 0.94 | 2.00 | 0.14 | 0.26 | 0.07 | 0.00 |
| BLAS 2; $y = A * x + y$ | DGEMV | 1.89 | | 0.29 | 0.42 | 0.15 | 0.03 |
| BLAS 3; $C = A * B + C$ | doloop(kji) | 6.29 | | 0.87 | 1.74 | 0.21 | 0.00 |
| BLAS 3; $C = A * B + C$ | DGEMM | 12.96 | | 1.84 | 1.26 | 0.59 | 0.01 |

Single CPU simple code hardware counter for simple math kernels

Processor AMD Budapest

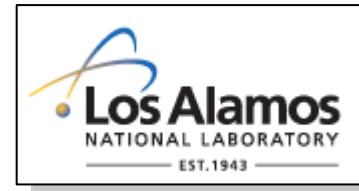
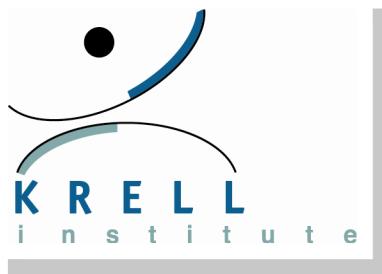
Other HWC metrics that are useful shown here

| code | 3dFFT; 256x256x256 | matmul 500x500 | QR Fact. N=2350 | HPCCG; sparseMV;100x100x100 |
|---------------------|--------------------|----------------|-----------------|-----------------------------|
| Comp. Inten;ops/ref | 1.33 | 1.71 | 1.68 | 0.64 |
| MFLOPS/papi | 952 | 4159 | 3738 | 352 |
| MFLOPS code | 1370 | 4187 | 4000 | 276 |
| percent peak | 19.8 | 86.7 | 77.9 | 7.3 |
| fpOps/TLB miss | 841.6515146 | 9040759.488 | 697703.964 | 14.05636016 |
| fpOps/D1 cache miss | 25.5290058 | 167.9364898 | 144.9081716 | 10.24364227 |
| fpOps/DC_MISS | 29.42427018 | 170.5178224 | 149.9578195 | 11.1702481 |
| ops/cycle | 0.4 | 1.75 | 1.56 | 0.15 |

Open | SpeedShop™

Section 4 I/O Tracing

SciDAC 2011 Tutorial
How to Analyze the Performance of Parallel Codes 101
A case study with Open|SpeedShop



- ❖ **I/O could be significant percentage of execution time dependent upon:**

- Checkpoint, analysis output, and visualization & I/O frequencies
- I/O pattern in the application:
 - N-to-1, N-to-N; simultaneous writes or requests
- Nature of application:
 - data intensive, traditional HPC, out-of-core
- File system and Striping: NFS, Lustre, Panasas, and #of OSTs
- I/O libraries: MPI-IO, hdf5, PLFS,...
- Other jobs stressing the I/O sub-systems

- ❖ **Obvious candidates to explore first while tuning:**

- Use parallel file system
- Optimize for I/O pattern
- Match checkpoint I/O frequency to MTBI of the system
- Use appropriate libraries

❖ Application: OOCORE benchmark from DOD HPCMO

- Out-of-core SCALPACK benchmark from UTK
- Can be configured to be disk I/O intensive
- Characterizes a very important class of HPC application involving the use of Method of Moments (MOM) formulation for investigating Electromagnetics (e.g. Radar Cross Section, Antenna design)
- Solves dense matrix equations by LU, QR or Cholesky
- Reference: *Benchmarking OOCORE, and out-of-core Matrix Solver, By Drs. Samuel B. Cable and Eduardo D'zevedo*

Why use this example?

- ❖ Used by HPCMO to evaluate I/O system scalability
- ❖ For our needs this application or similar out-of-core dense solver benchmarks help to point out importance of the following in performance analysis
 - I/O overhead minimization
 - Matrix Multiply kernel – possible to achieve close to peak performance of the machine if tuned well
 - “blocking” very important to tune for deep memory hierarchies

Use OSS to measure and tune for I/O

INPUT: testdriver.in

ScalAPACK out-of-core LU,QR,LL
factorization input file

testdriver.out

6 device out

1 number of factorizations

LU factorization methods -- QR, LU,
or LT

1 number of problem sizes

31000 values of M

31000 values of N

1 values of nrhs

9200000 values of Asize

1 number of MB's and NB's

16 values of MB

16 values of NB

1 number of process grids

4 values of P

4 values of Q

Run on 16 cores on an SNL Quad-Core, Quad-Socket Opteron IB Cluster

Investigate File system impact with OpenSpeedShop: Compare
Lustre I/O with striping to NFS I/O

run cmd: ossio “srun -N 1-n 16 ./testzdriver-std”

Sample Output from Lustre run:

TIME M N MB NB NRHS P Q Fact/SolveTime Error Residual

WALL 31000 31000 16 16 1 4 4 1842.20 1611.59 4.51E+15
1.45E+11

DEPS = 1.110223024625157E-016

sum(xsol_i) = (30999.999999873,0.000000000000000E+000)

sum |xsol_i - x_i| =
(3.332285336962339E-006,0.000000000000000E+000)

sum |xsol_i - x_i|/M =
(1.074930753858819E-010,0.000000000000000E+000)

sum |xsol_i - x_i|/(M*eps) =
(968211.548505533,0.000000000000000E+000)

From output of two separate runs using Lustre and NFS:

LU Fact time with Lustre= 1842 secs;

LU Fact time with NFS = 2655 secs

813 sec penalty (more than 30%) if you do not use parallel file
system like Lustre!

NFS and Lustre OSS Analysis (screen shot from Lustre)

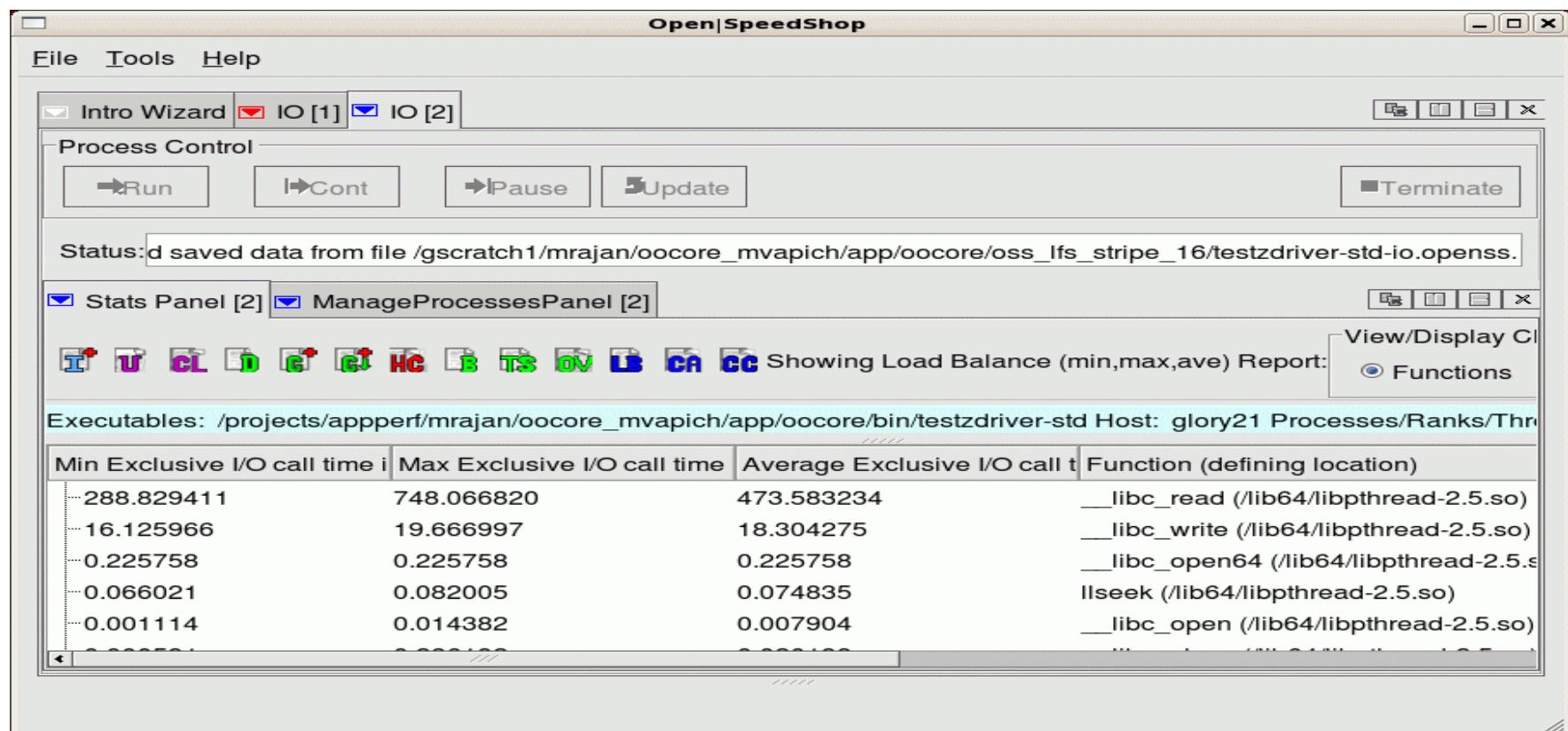
The run time difference 75% of 813 secs is mostly I/O: $(1360+99) - (847 + 7) = 605$ secs

NFS RUN

| Min t (secs) | Max t (secs) | Avg t (secs) | call Function |
|--------------|--------------|--------------|---|
| 1102.380076 | 1360.727283 | 1261.310157 | <code>__libc_read(/lib64/libpthread-2.5.so)</code> |
| 31.19218 | 99.444468 | 49.01867 | <code>__libc_write(/lib64/libpthread-2.5.so)</code> |

LUSTRE RUN

| Min t (secs) | Max t (secs) | Avg t (secs) | call Function |
|--------------|--------------|--------------|---|
| 368.898283 | 847.919127 | 508.658604 | <code>__libc_read(/lib64/libpthread-2.5.so)</code> |
| 6.27036 | 7.896153 | 6.850897 | <code>__libc_write(/lib64/libpthread-2.5.so)</code> |



Lustre File System (lfs) commands:

lfs setstripe -s (size bytes; k, M, G) -c (count; -1 all) -I (index; -1 round robin) <file | directory>

Typical defaults: -s 1M -c 4 -I -1 (usually good to try first)

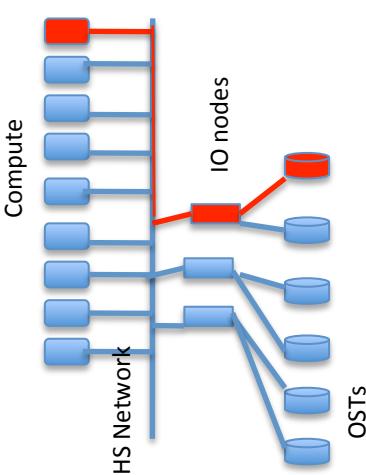
File striping is set upon file creation.

lfs getstripe <file | directory>

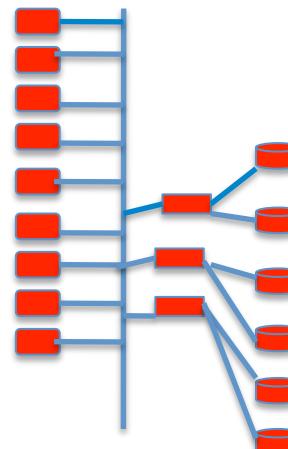
Example: *lfs getstripe --verbose ./oss_lfs_stripe_16 | grep stripe_count*

stripe_count: 16 stripe_size: 1048576 stripe_offset: -1

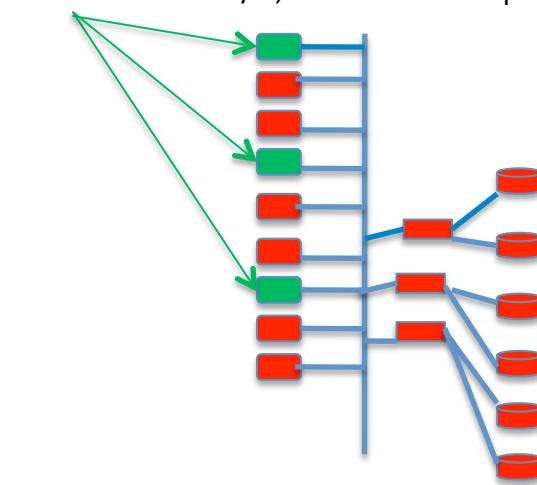
1 PE writes; BW limited



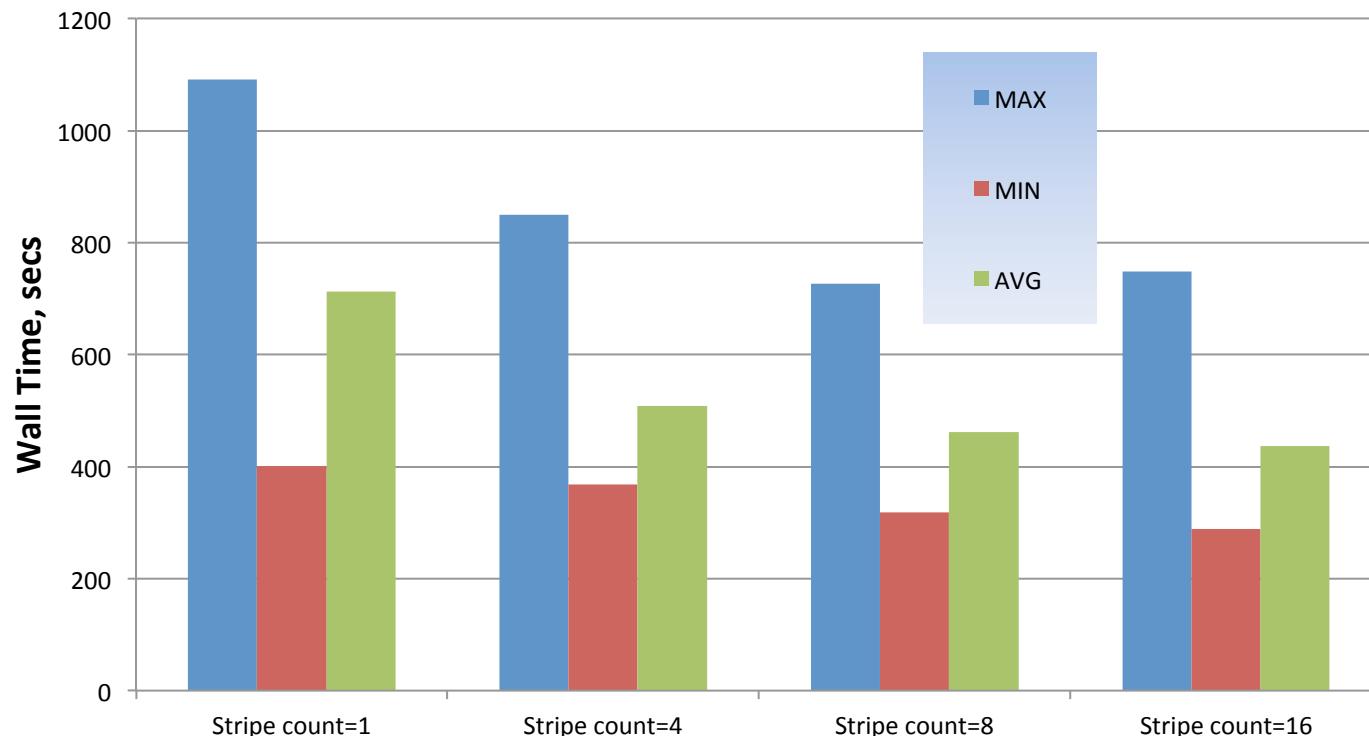
1 file per process; BW enhanced



Subset of PEs do I/O; Could be most optimal

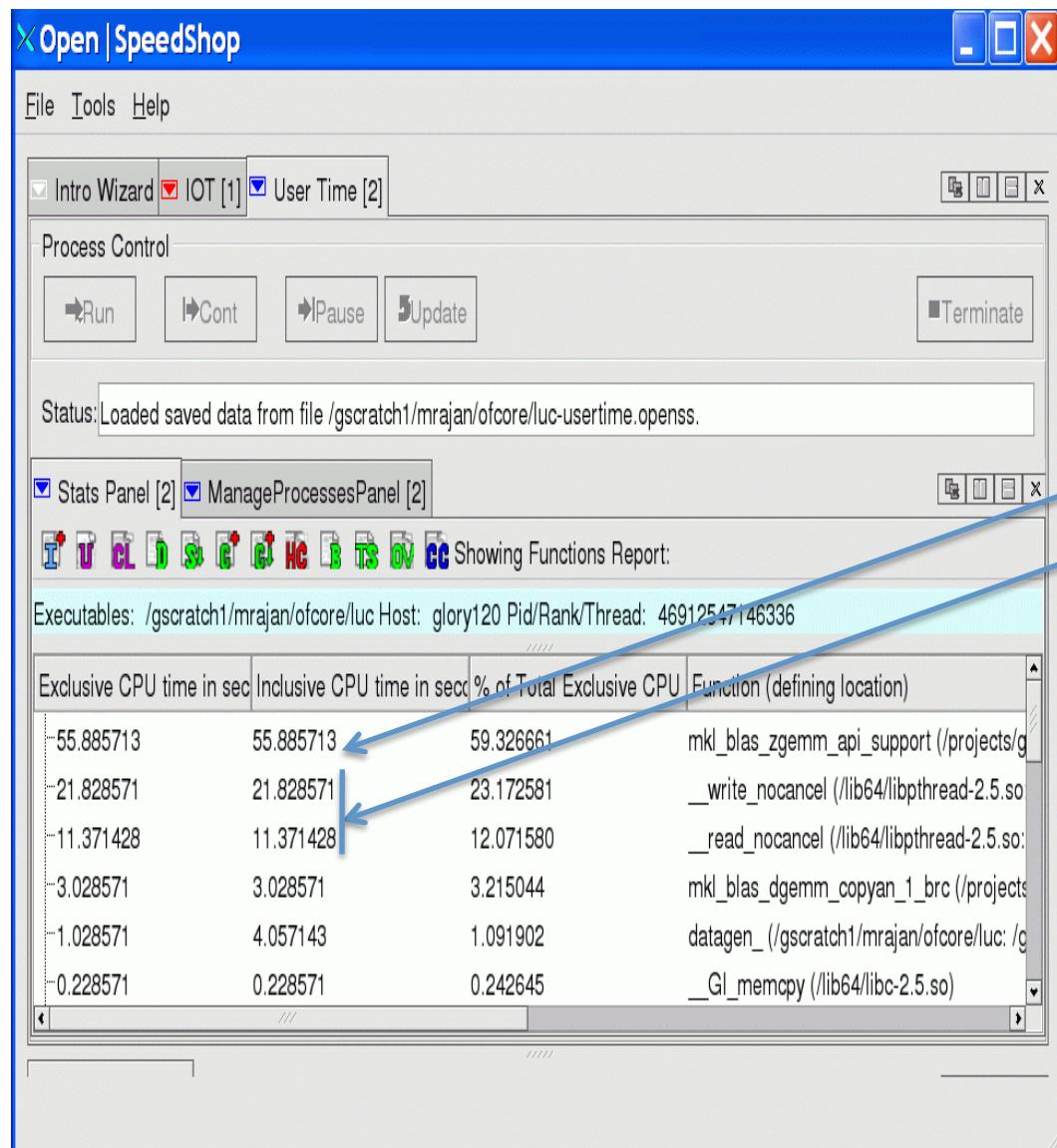


OOCORE I/O performance; libc_read time from OpenSpeedShop



Use OpenSpeedShop *usertime* experiment to profile;

This example compares OSS data to code instrumentation data(yours may not be instrumented!)



OutPut from code:

***** PROBLEM PARAMETERS *****

PROBLEM SIZE = 5000 BLOCK SIZE 100
NUMBER OF RIGHT HAND SIDES 100
SUB-BLOCK SIZE 50

***** PERFORMANCE TIME IN SECONDS *

total time 104.3114 seconds
factor time 100.2688 seconds
backsolve/FR time 2.075548 seconds
matmul time 53.80486 seconds
io time 45.86675 seconds
inverse time 6.3550234E-02 seconds

lu diag. blk. time 9.2301369E-02 seconds

triangular solver time 1.932023 seconds

***** PERFORMANCE FIGURES FOR SOLVER *****

NUMBER OF MATRIX MULTIPLICATIONS 45425.00
NUMBER OF DISK READ/WRITES 125050.0
NUMBER OF FLOATING POINT OPERATIONS= 3.3353335E+11

MFLOPS FOR FACTORING 3324.399
OVERALL MFLOPS 3197.479
MATRIX MULITPLY MFLOPS 6194.608
INVERSION MFLOPS 1573.558
DISK TRANSFER RATE 436.2202 MBYTES/SEC

❖ Extended I/O Tracing (iot experiment)

- Records each event in chronological order
- Collects Additional Information
 - Function Parameters
 - Function Return Value

❖ When to use extended I/O tracing?

- When you want to trace the exact order of events
- When you want to see the return values or bytes read or written.

Beware of Serial I/O in applications: Encountered in VOSS, code LeP: Simple code here illustrates (acknowledgment: Mike Davis, Cray Inc)



```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define VARS_PER_CELL 15

/***
 * Write a single restart file from many MPI processes */
int
write_restart (
    MPI_Comm comm           /// MPI communicator
, int num_cells           /// number of cells on this process
, double *cellv            /// cell vector
) {

    int rank;                // rank of this process within comm
    int size;                // size of comm
    int tag;                 // for MPI_Send, MPI_Recv
    int baton;               // for serializing I/O
    FILE *f;                 // file handle for restart file

    /**
     * Procedure:
     * Get MPI parameters
     */

    MPI_Comm_rank (comm, &rank);
    MPI_Comm_size (comm, &size);
    tag = 4747;
```

```
if (rank == 0) {
    /**
     * Rank 0 create a first restart file,
     * and start the serial I/O;
     * write cell data, then pass the baton to rank 1
     */

    f = fopen ("restart.dat", "wb");
    fwrite (cellv, num_cells, VARS_PER_CELL * sizeof (double), f);
    fclose (f);
    MPI_Send (&baton, 1, MPI_INT, 1, tag, comm);
} else {

    /**
     * Ranks 1 and higher wait for previous rank to complete I/O,
     * then append its cell data to the restart file,
     * then pass the baton to the next rank
     */

    MPI_Recv (&baton, 1, MPI_INT, rank - 1, tag, comm, MPI_STATUS_IGNORE);
    f = fopen ("restart.dat", "ab");
    fwrite (cellv, num_cells, VARS_PER_CELL * sizeof (double), f);
    fclose (f);
    if (rank < size - 1) {
        MPI_Send (&baton, 1, MPI_INT, rank + 1, tag, comm);
    }
}

/**
 * All ranks have posted to the restart file;
 * return to caller
 */

return 0;
}
```

```
int main (int argc, char *argv[]) {
    MPI_Comm comm;
    int comm_rank;
    int comm_size;
    int num_cells;
    double *cellv;
    int i;

    MPI_Init (&argc, &argv);

    MPI_Comm_dup (MPI_COMM_WORLD, &comm);
    MPI_Comm_rank (comm, &comm_rank);
    MPI_Comm_size (comm, &comm_size);
    /**
     * Make the cells be distributed somewhat evenly across ranks
     */
    num_cells = 5000000 + 2000 * (comm_size / 2 - comm_rank);
    cellv = (double *) malloc (num_cells * VARS_PER_CELL * sizeof (double));
    for (i = 0; i < num_cells * VARS_PER_CELL; i++) {
        cellv[i] = comm_rank;
    }
    write_restart (comm, num_cells, cellv);
    MPI_Finalize ();
    return 0;
}
```

IOT O|SS Experiment of Serial I/O Example

Open | SpeedShop

File Tools Help

Custom Experiment [1]

Process Control

Run Cont Pause Update Terminate

Status: Process Loaded: Click on the "Run" button to begin the experiment.

Stats Panel [1] ManageProcessesPanel [1]

I U CL D G Hc B TS DV EL LB CA CC Showing Per Event Report: View/Display Choice Functions

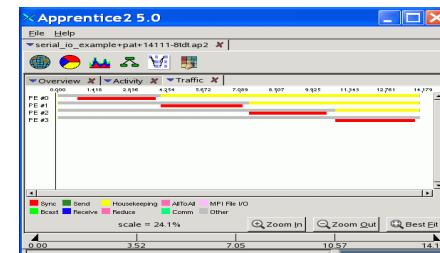
Executables: (none) Host: glory238 Processes/Ranks/Threads:(4) 0 ...

| Start Time | I/O Call Time(ms) | % of Total Time | Call Stack Function (defining location) |
|---------------------|-------------------|-----------------|---|
| 2010/09/08 13:22:54 | 0.029000 | 6.682028 | >_libc_read (/lib64/libpthread-2.5.so) |
| 2010/09/08 13:22:54 | 0.026000 | 5.990783 | >_libc_write (/lib64/libpthread-2.5.so) |
| 2010/09/08 13:22:54 | 0.008000 | 1.843318 | >_libc_read (/lib64/libpthread-2.5.so) |
| 2010/09/08 13:22:54 | 0.058000 | 13.364055 | >_libc_write (/lib64/libpthread-2.5.so) |
| 2010/09/08 13:22:54 | 0.061000 | 14.055300 | >_libc_write (/lib64/libpthread-2.5.so) |
| 2010/09/08 13:22:54 | 0.010000 | 2.304147 | >_libc_read (/lib64/libpthread-2.5.so) |
| 2010/09/08 13:22:54 | 0.016000 | 3.686636 | >_libc_read (/lib64/libpthread-2.5.so) |
| 2010/09/08 13:22:54 | 0.015000 | 3.456221 | >_libc_read (/lib64/libpthread-2.5.so) |
| 2010/09/08 13:22:54 | 0.025000 | 5.760369 | >_libc_read (/lib64/libpthread-2.5.so) |
| 2010/09/08 13:22:54 | 0.021000 | 4.838710 | >_libc_read (/lib64/libpthread-2.5.so) |
| 2010/09/08 13:22:54 | 0.015000 | 3.456221 | >_libc_write (/lib64/libpthread-2.5.so) |

SHOWS EVENT BY EVENT LIST:

Clicking on this gives each call to a I/O function being traced as shown.

Below is a graphical trace view of the same data showing serialization of **fwrite()** (THE RED BARS for each PE) with another tool.



Offline io/iot experiment on smg2000 application

Convenience script basic syntax

ossio[t] "smg2000 -n 50 50 50" [default | <list of I/O func>]

➤ Parameters

- I/O Function list to trace (default is all)
- creat, creat64, dup, dup2, lseek, lseek64, open, open64, pipe, pread, pread64, pwrite, pwrite64, read, readv, write, writev

❖ Extended I/O Tracing (iot experiment)

- Records each event in chronological order
- Collects Additional Information
 - Function Parameters
 - Function Return Value

❖ When to use extended I/O tracing?

- When you want to trace the exact order of events
- When you want to see the return values or bytes read or written.

I/O output via CLI (equivalent of HC in GUI)

openss>>expview -vcalltrees,fullstack iot1

| I/O Call Time(ms) | % of Total Time | Number of Calls | Call Stack Function (defining location) |
|-------------------|-----------------|-----------------|---|
|-------------------|-----------------|-----------------|---|

_start (sweep3d.mpi)

> @ 470 in __libc_start_main (libmonitor.so.0.0.0: main.c,450)

>>__libc_start_main (libc-2.10.2.so)

>>> @ 428 in monitor_main (libmonitor.so.0.0.0: main.c,412)

>>>>main (sweep3d.mpi)

>>>>> @ 58 in MAIN__ (sweep3d.mpi: driver.f,1)

>>>>>> @ 25 in task_init_ (sweep3d.mpi: mpi_stuff.f,1)

>>>>>>_gfortran_ftell_i2_sub (libgfortran.so.3.0.0)

>>>>>>_gfortran_ftell_i2_sub (libgfortran.so.3.0.0)

....

>>>>>>>>>_gfortran_st_read (libgfortran.so.3.0.0)

| | | |
|--------------|--------------|---|
| 17.902981000 | 96.220812461 | 1 >>>>>>>>>__libc_read (libpthread-2.10.2.so) |
|--------------|--------------|---|

Section Summary / I/O Tradeoffs

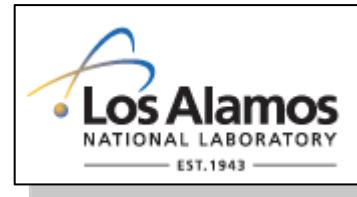
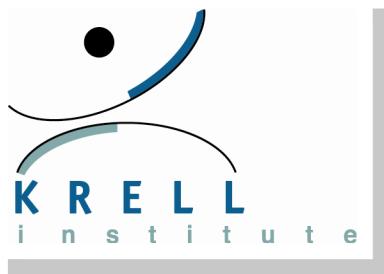
- ❖ **Avoid writing to one file from all MPI tasks.**
 - If you need to do it make sure distinct offsets for each PE starts at a stripe boundary. Use Buffered IO if you must.
- ❖ **If each process writes it's own file then the parallel file system attempts to load balance the OST taking advantage of the stripe characteristics**
- ❖ **Meta data server overhead can often create severe I/O problems.**
 - Minimize number of files accessed per PE and minimize each PE doing operations like *seek*, *open*, *close*, *stat* that involve inode information
- ❖ **I/O time is usually not measured even in applications that keep some function profile**
 - Open|SpeedShop can shed light on time spent in I/O using **io**, **iot** experiments

Open | SpeedShop™

Section 6 How to Detect Bottlenecks in Parallel Codes

SciDAC 2011 Tutorial
How to Analyze the Performance of Parallel Codes 101

A case study with Open|SpeedShop



❖ Ideal scenario

- Efficient threading when using pthreads or OpenMP
 - All threads are assigned work that can execute concurrently
- Load balance for parallel jobs using MPI
 - All MPI ranks doing same amount of work, so no MPI rank waits

❖ What causes the ideal goal to fail?*

- Equal work was not given to each rank
 - Number of array operations not equal for each rank
 - Loop iterations not evenly distributed
- Sometimes have issues even if data is distributed equally
 - Sparse arrays: some ranks have work others have 0's
 - Adaptive grid models: some ranks need to redefine their mesh while others don't
 - N-body simulations: some data/work migrates to another ranks domain, so that rank has more work

*From LLNL parallel processing tutorial

❖ O|SS is designed to work on parallel jobs

- Support for threading and message passing
- Automatically tracks all ranks and threads during execution
- Records/stores performance info per process/rank/thread

❖ All experiments can be used on parallel jobs

- O|SS applies collector to all ranks on all nodes
- Result display:
 - Default: Aggregate results across all ranks and/or threads
 - Optional: Select individual or groups of ranks or threads

❖ MPI specific tracing experiments

- Tracing of MPI function calls (individual, all, or a specific group)
- Three forms of MPI tracing experiments

❖ O|SS has been tested with a variety of MPIs

- Including: Open MPI, MVAPICH[2], and MPICH2 (Intel, Cray)

❖ Identifying MPI tasks (How we know the rank info)

- Online version: through MPIR interface
- Offline version: through PMPI preload

❖ Running O|SS experiments on MPI codes

- Add MPI starter/driver phrase as part of executable name
- ossmpi “mpirun –np 128 sweep3d.mpi”
- osspcsamp “mpirun –np 32 sweep3d.mpi”
- ossio “srun –N 4 –n 16 sweep3d.mpi”
- openss –offline –f “mpirun –np 128 sweep3d.mpi” hwctime
- openss –online –f “srun –N 8 –n 128 sweep3d.mpi” usertime

❖ Default views

- Values aggregated across all ranks
- Manually include/exclude individual ranks/processes/threads

❖ Comparing ranks

- Use the Customize Stats Panel View
- Create compare columns for process groups or individual ranks

❖ Cluster analysis (finding outliers)

- Automatically creates groups of similar performing ranks or threads
- Available from the Stats Panel toolbar or context menu

❖ O|SS provides common analysis functions

- Designed for quick analysis of MPI applications
- Create new views in the StatsPanel
- Accessible through the context menu or toolbar

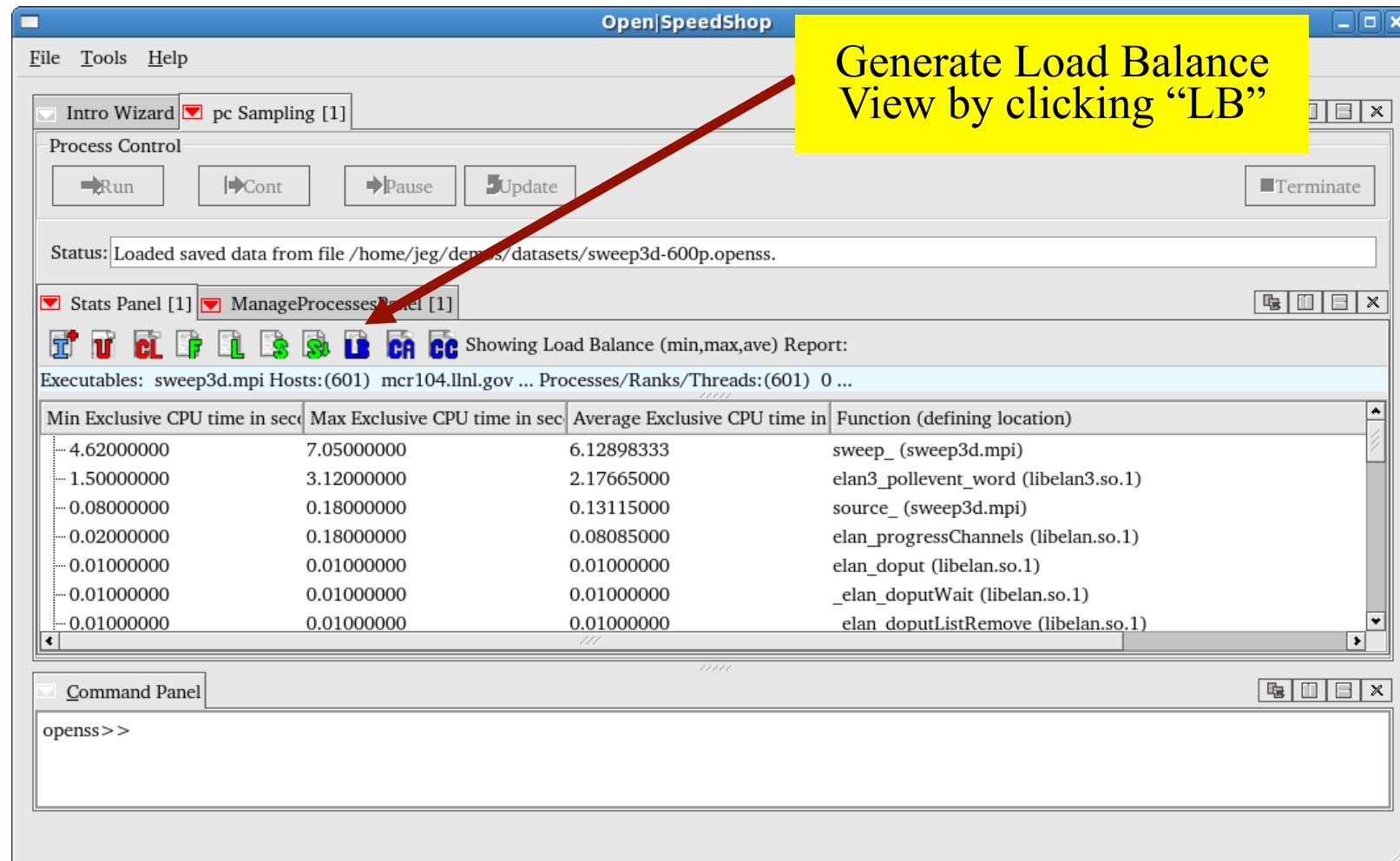
❖ Load Balance View

- Calculates min, max, average across ranks, processes, or threads

❖ Comparative Analysis View

- Uses “cluster analysis” algorithm to group similar performing ranks, processes, or threads

❖ Min, Max, Average across Ranks/Threads/Processes



Open|SpeedShop

File Tools Help

Intro Wizard pc Sampling [1]

Process Control

Run Cont Pause Update Terminate

Status: Loaded saved data from file /home/jeg/demos/datasets/sweep3d-600p.openss.

Stats Panel [1] ManageProcessesPanel [1]

T U CL F L S S LB CA CC Showing Load Balance (min,max,ave) Report:

Executables: sweep3d.mpi Hosts:(601) mcr104.llnl.gov ... Processes/Ranks/Threads:(601) 0 ...

| Min Exclusive CPU time in sec | Max Exclusive CPU time in sec | Average Exclusive CPU time in sec | Function (defining location) |
|-------------------------------|-------------------------------|-----------------------------------|--------------------------------------|
| -4.62000000 | 7.05000000 | 6.12898333 | sweep_ (sweep3d.mpi) |
| -1.50000000 | 3.12000000 | 2.17665000 | elan3_pollevent_word (libelan3.so.1) |
| -0.08000000 | 0.18000000 | 0.13115000 | source_ (sweep3d.mpi) |
| -0.02000000 | 0.18000000 | 0.08085000 | elan_progressChannels (libelan.so.1) |
| -0.01000000 | 0.01000000 | 0.01000000 | elan_doput (libelan.so.1) |
| -0.01000000 | 0.01000000 | 0.01000000 | _elan_doputWait (libelan.so.1) |
| -0.01000000 | 0.01000000 | 0.01000000 | elan_doputListRemove (libelan.so.1) |

Command Panel

openss>>

Generate Load Balance View by clicking "LB"

Comparative Analysis View (clustering)

❖ Group like performing Ranks/Threads/Processes

Open|SpeedShop

File Tools Help

Intro Wizard pc Sampling [1]

Process Control

Run Cont Pause Update Terminate

Status: Loaded saved data from file /home/jeg/demos/datasets/sweep3d-600p.openss.

Stats Panel [1] ManageProcessesPanel [1]

I U CL F L S S LB CA CC Showing Comparative Analysis Report:

Executables: sweep3d.mpi

View consists of comparison columns click on the metadata icon for details.

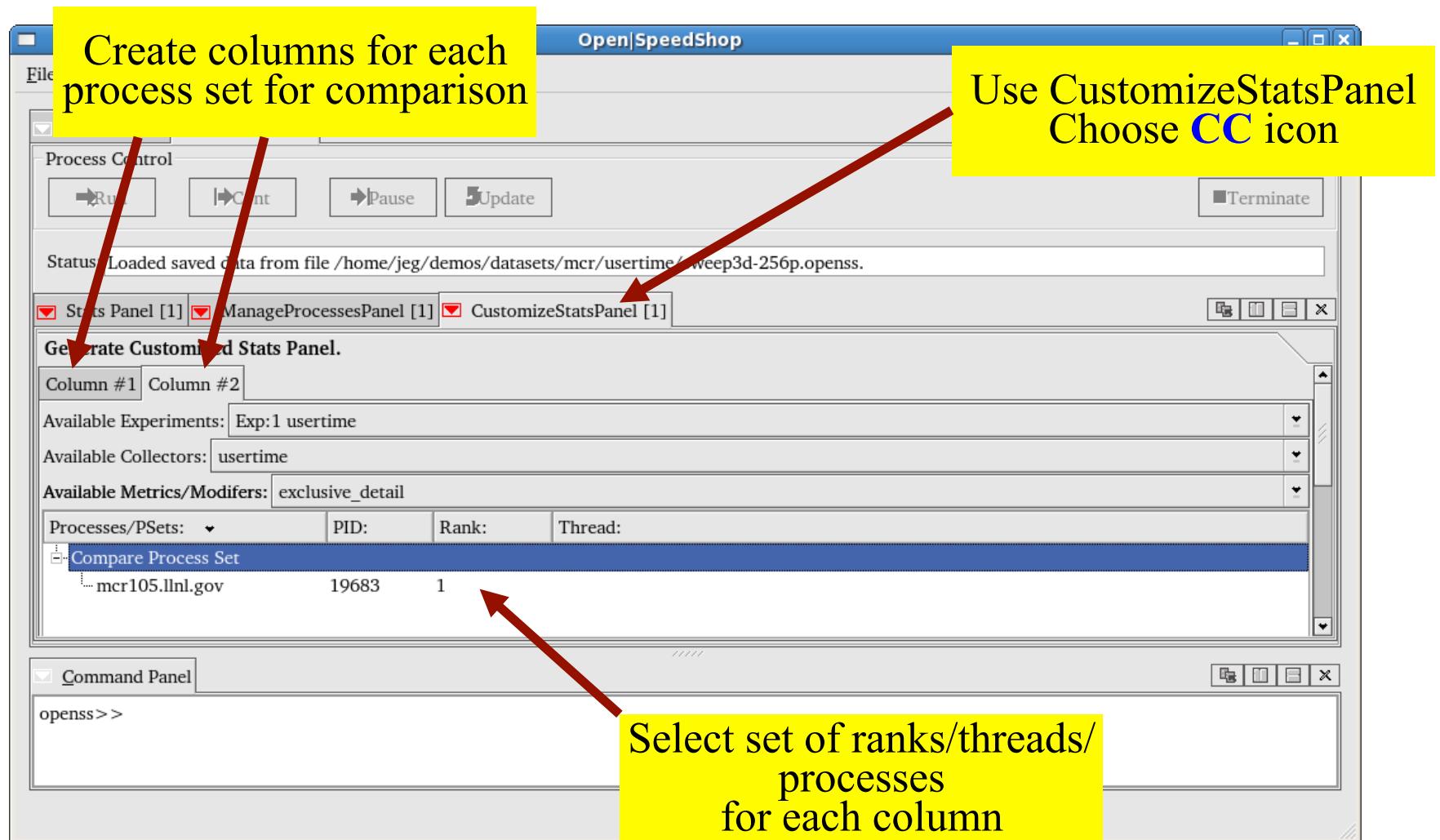
| pcsample -x 1 -r 0:34, 36:91, 93 | pcsample -x 1 -r 35, 92, 160, 40 | pcsample -x 1 -r 448, Average E | Function (defining location) |
|----------------------------------|----------------------------------|---------------------------------|---------------------------------------|
| ... 6.25470930 | 5.36554217 | 4.62000000 | sweep_ (sweep3d.mpi) |
| ... 2.08813953 | 2.71554217 | 3.12000000 | elan3_pollevent_word (libelan3.so.1) |
| ... 0.13131783 | 0.12987952 | 0.15000000 | source_ (sweep3d.mpi) |
| ... 0.07777132 | 0.09939759 | 0.13000000 | elan_progressChannels (libelan.so.1) |
| ... 0.06873786 | 0.09325301 | 0.16000000 | elan_pollWord (libelan.so.1) |
| ... 0.06025243 | 0.07120482 | 0.07000000 | elan_progressFragLists (libelan.so.1) |

Command Panel

openss >

Generate Cluster Analysis by clicking "CA"

Comparing Ranks (1)



Comparing Ranks (2)

Open|SpeedShop

File Tools Help

Intro Wizard User Time [1]

Process Control

Rank 0 Rank 1

Status: Loaded saved data from file /home/jeg/demos/datasets/mcr/usertime/sweep3d-256p.openss.

Stats Panel [1] ManageProcessesPanel [1] CustomizeStatsPanel [1]

I U CL F L S S G G G T T T T T T B L B CA CC Showing Functions Report:

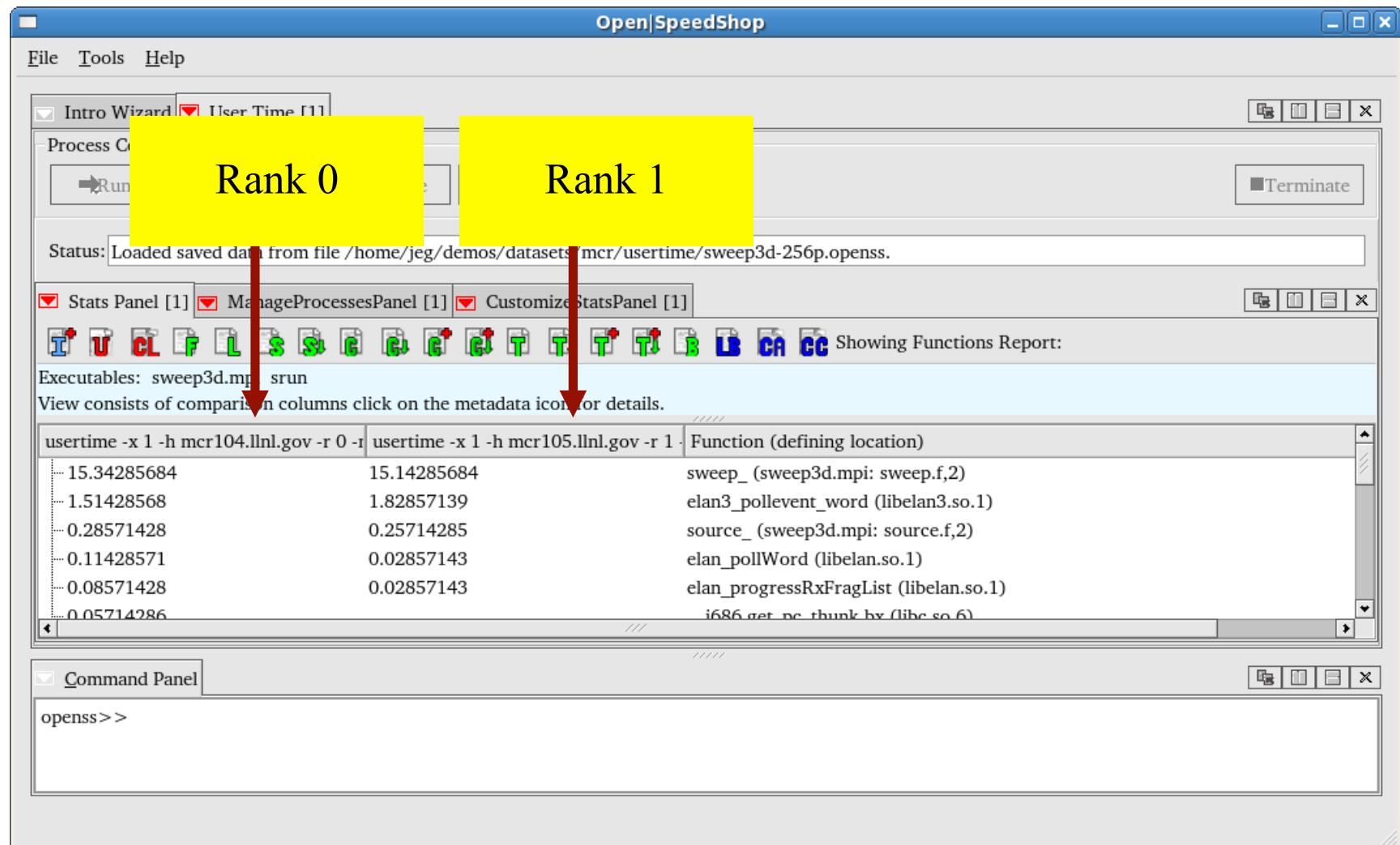
Executables: sweep3d.mpi srun

View consists of comparison columns click on the metadata icon for details.

| usertime -x 1 -h mcr104.llnl.gov -r 0 -i | usertime -x 1 -h mcr105.llnl.gov -r 1 -i | Function (defining location) |
|--|--|--|
| 15.34285684 | 15.14285684 | sweep_ (sweep3d.mpi: sweep.f,2) |
| 1.51428568 | 1.82857139 | elan3_pollevent_word (libelan3.so.1) |
| 0.28571428 | 0.25714285 | source_ (sweep3d.mpi: source.f,2) |
| 0.11428571 | 0.02857143 | elan_pollWord (libelan.so.1) |
| 0.08571428 | 0.02857143 | elan_progressRxFragList (libelan.so.1) |
| 0.05714286 | | i686_get_pc_thunk_bv (libc.so.6) |

Command Panel

```
openss>>
```



❖ Similar to I/O tracing

- Record all MPI call invocations
- By default: record call times (mpi)
 - Convenience script: ossmpi
- Optional: record call times and arguments (mpit)
 - Convenience script: ossmpit

❖ Equal events will be aggregated

- Save space in O|SS database
- Reduces overhead

❖ Public format:

- Full MPI traces in Open Trace Format (OTF)
- Experiment name: (mpiotf)
 - Convenience script: ossmpiotf

Offline mpi/mpit/mpiof experiment on smg2000 appl.

Convenience script basic syntax

```
ossmpi[t] "srun -N 4 -n 32 smg2000 -n 50 50 50" [default | <list MPI func> | mpi category]
```

➤ Parameters

- Default is all MPI Functions Open|SpeedShop traces
- MPI Function list to trace (comma separated)
 - MPI_Send, MPI_Recv,
- mpi_category:
 - "all", "asynchronous_p2p", "collective_com", "datatypes", "environment", "graphs_contexts_comms", "persistent_com", "process_topologies", "synchronous_p2p"

MPI Tracing Results: Default View

Open|SpeedShop

File Tools Help

MPI [1]

Process Control

Run Cont Pause Terminate

Status: Process Loader.. Click on the "Run" button to begin the experiment.

Stats Panel [1] ManageProcessesPanel [1]

Showing Functions Report

Aggregated Results

Executables: smg2000 Hosts:(64) hyperion583.llnl.gov ... Processes/Ranks/Threads:(512) 0 ...

Metadata for Experiment 1:

Application command:
Executables: smg2000
Experiment type: mpi
Host(s): hyperion583.llnl.gov hyperion584.llnl.gov hyperion585.llnl.gov hyperion586.llnl.gov hyperion587.llnl.gov hyperion588.llnl.gov hyperion589.llnl.gov h
Processes, Ranks or Threads: 0-511

| Minimum MPI Call Time(ms) | Maximum MPI Call Time(ms) | Average Time(ms) | Number of Calls | Function (defining location) |
|---------------------------|---------------------------|------------------|-----------------|---|
| - 555.306000 | 1276.275000 | 755.289027 | 512 | PMPI_Init (libmonitor.so.0.0: pmpi.c,94) |
| - 151.147000 | 167.504000 | 163.231094 | 512 | PMPI_Finalize (libmonitor.so.0.0: pmpi.c,223) |
| - 0.152000 | 0.474000 | 0.334205 | 512 | MPI_Allgatherv (libmpich.so.1.0: allgatherv.c,73) |
| - 0.043000 | 0.212000 | 0.133098 | 512 | MPI_Allgather (libmpich.so.1.0: allgather.c,70) |
| - 0.031000 | 2.034000 | 1.312102 | 512 | MPI_Barrier (libmpich.so.1.0: barrier.c,56) |
| - 0.013000 | 10.322000 | 0.717578 | 6144 | MPI_Allreduce (libmpich.so.1.0: allreduce.c,59) |
| - 0.000001 | 611.617000 | 0.977852 | 4667648 | MPI_Waitall (libmpich.so.1.0: waitall.c,57) |
| - 0.000001 | 0.600000 | 0.001156 | 5403936 | MPI_Isend (libmpich.so.1.0: isend.c,58) |
| - 0.000001 | 0.069000 | 0.000665 | 5403936 | MPI_Irecv (libmpich.so.1.0: irecv.c,48) |

View Results: Show MPI Callstacks

Open|SpeedShop

File Tools

MPI [1] Process Control

Run

Unique Call Paths View:
Click C+ Icon

Status: Process Loaded: Click on the "Run" button to begin the experiment.

Stats Panel [1] ManageProcessesPanel [1]

I U CL D G GT HC B TS OV LB CA CC Showing CallTrees,FullStack Report:

Executables: smg2000 Host: localhost.localdomain Processes/Ranks/Threads:(2) 0 ...

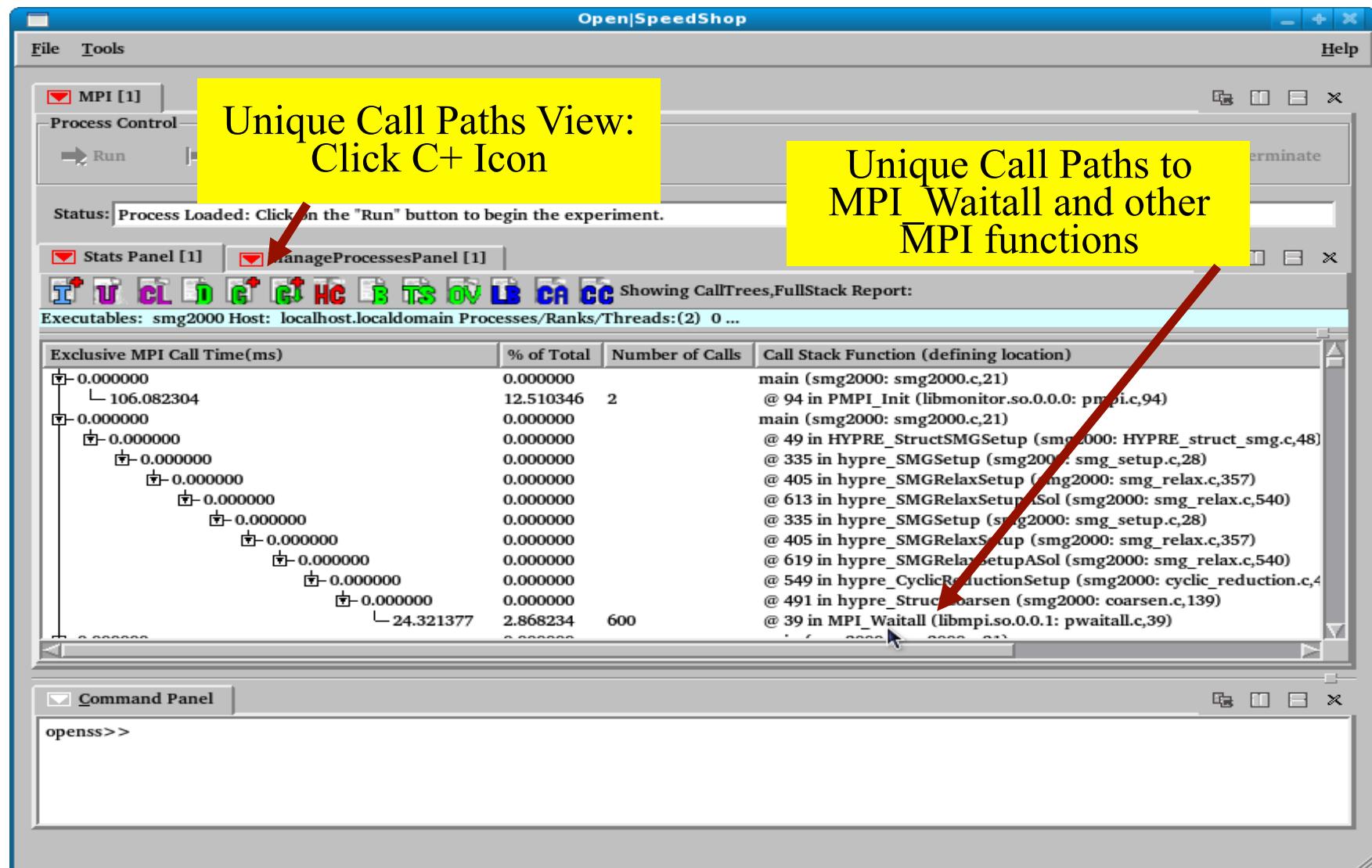
Exclusive MPI Call Time(ms)

| | % of Total | Number of Calls | Call Stack Function (defining location) |
|------------|------------|---|---|
| 0.000000 | 0.000000 | main (smg2000: smg2000.c,21) | |
| 106.082304 | 12.510346 | 2 @ 94 in PMPI_Init (libmonitor.so.0.0.0: pmpi.c,94) | |
| 0.000000 | 0.000000 | main (smg2000: smg2000.c,21) | |
| 0.000000 | 0.000000 | @ 49 in HYPRE_StructSMGSetup (smg2000: HYPRE_struct_smg.c,48) | |
| 0.000000 | 0.000000 | @ 335 in hypre_SMGSetup (smg2000: smg_setup.c,28) | |
| 0.000000 | 0.000000 | @ 405 in hypre_SMGRelaxSetup (smg2000: smg_relax.c,357) | |
| 0.000000 | 0.000000 | @ 613 in hypre_SMGRelaxSetupASol (smg2000: smg_relax.c,540) | |
| 0.000000 | 0.000000 | @ 335 in hypre_SMGSetup (smg2000: smg_setup.c,28) | |
| 0.000000 | 0.000000 | @ 405 in hypre_SMGRelaxSetup (smg2000: smg_relax.c,357) | |
| 0.000000 | 0.000000 | @ 619 in hypre_SMGRelaxSetupASol (smg2000: smg_relax.c,540) | |
| 0.000000 | 0.000000 | @ 549 in hypre_CyclicReductionSetup (smg2000: cyclic_reduction.c,4) | |
| 0.000000 | 0.000000 | @ 491 in hypre_StructCoarsen (smg2000: coarsen.c,139) | |
| 24.321377 | 2.868234 | @ 39 in MPI_Waitall (libmpi.so.0.0.1: pwaitall.c,39) | |

Unique Call Paths to
MPI_Waitall and other
MPI functions

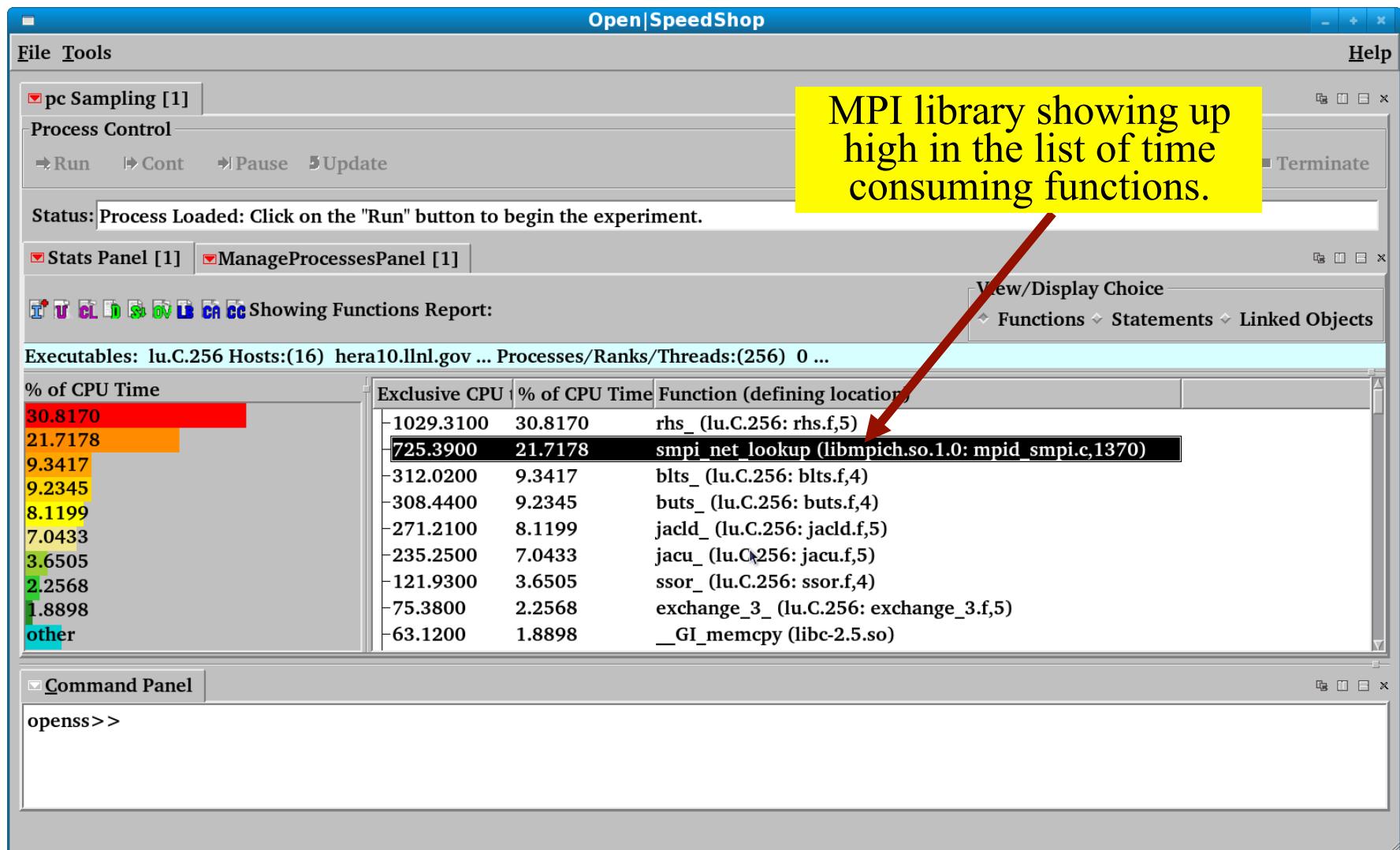
Command Panel

openss>>



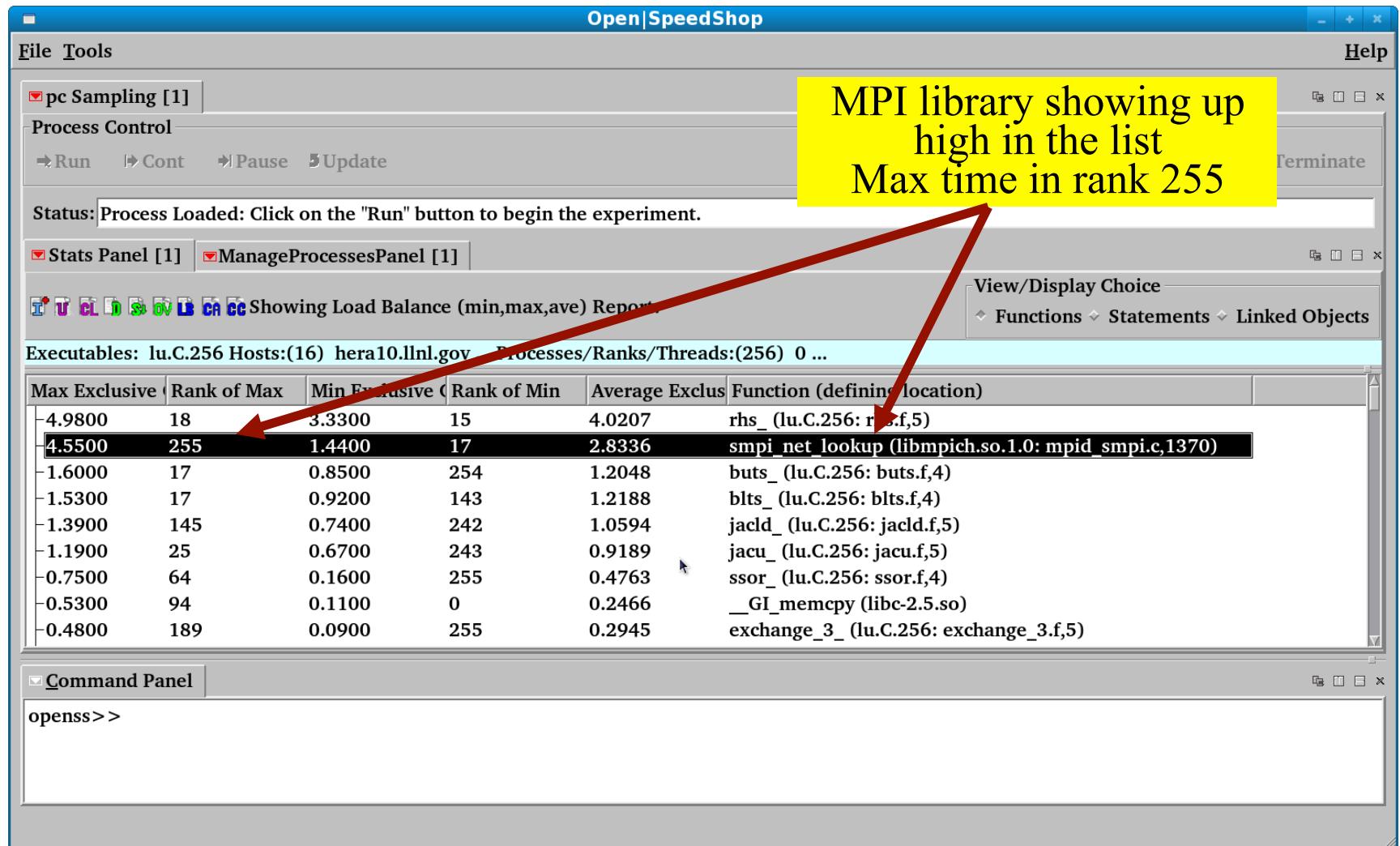
A MPI Case Study: Using NPB: LU

❖ Default pcsamp view based on functions



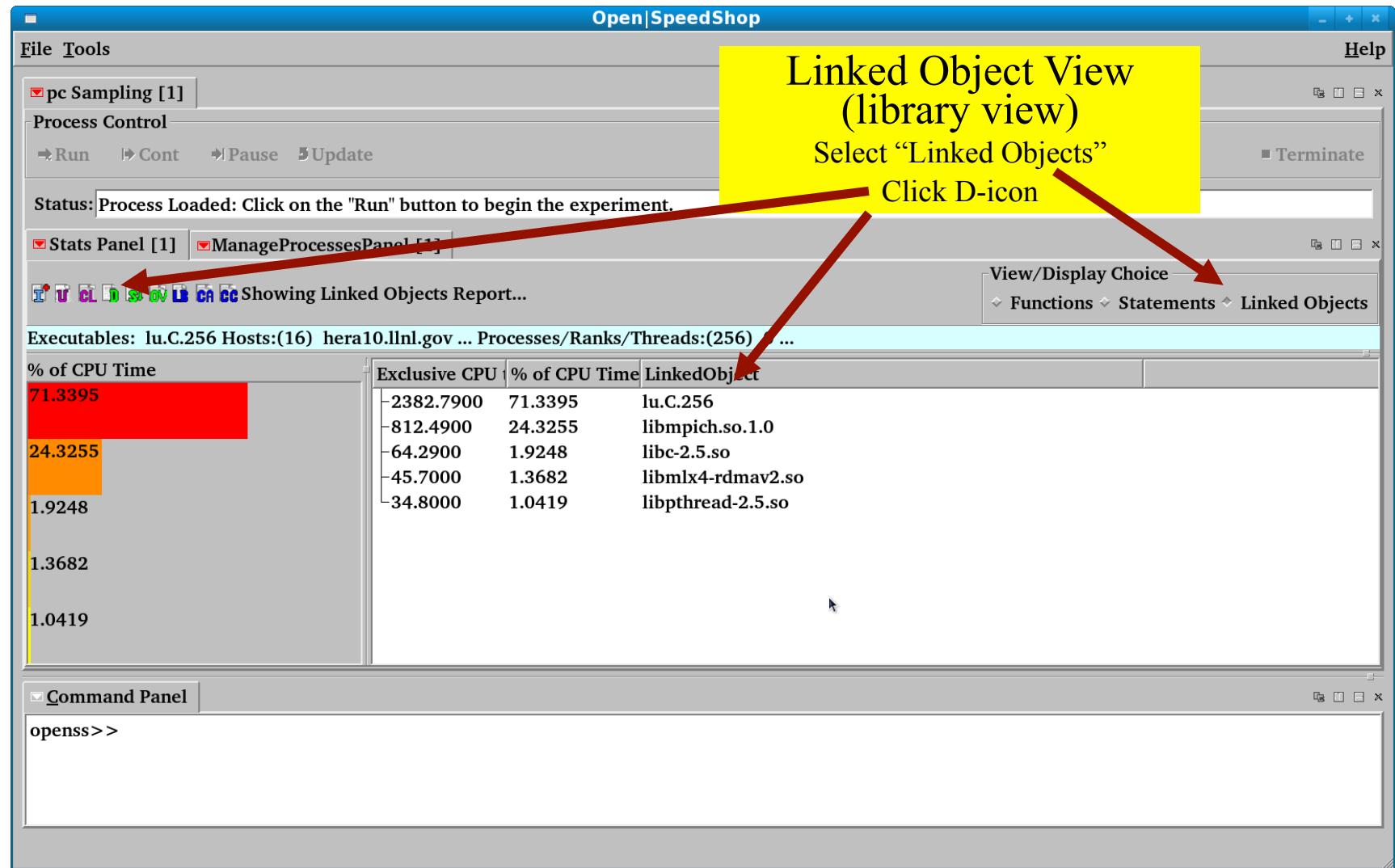
A MPI Case Study: Using NPB: LU

❖ Load Balance View based on functions (pcsample)



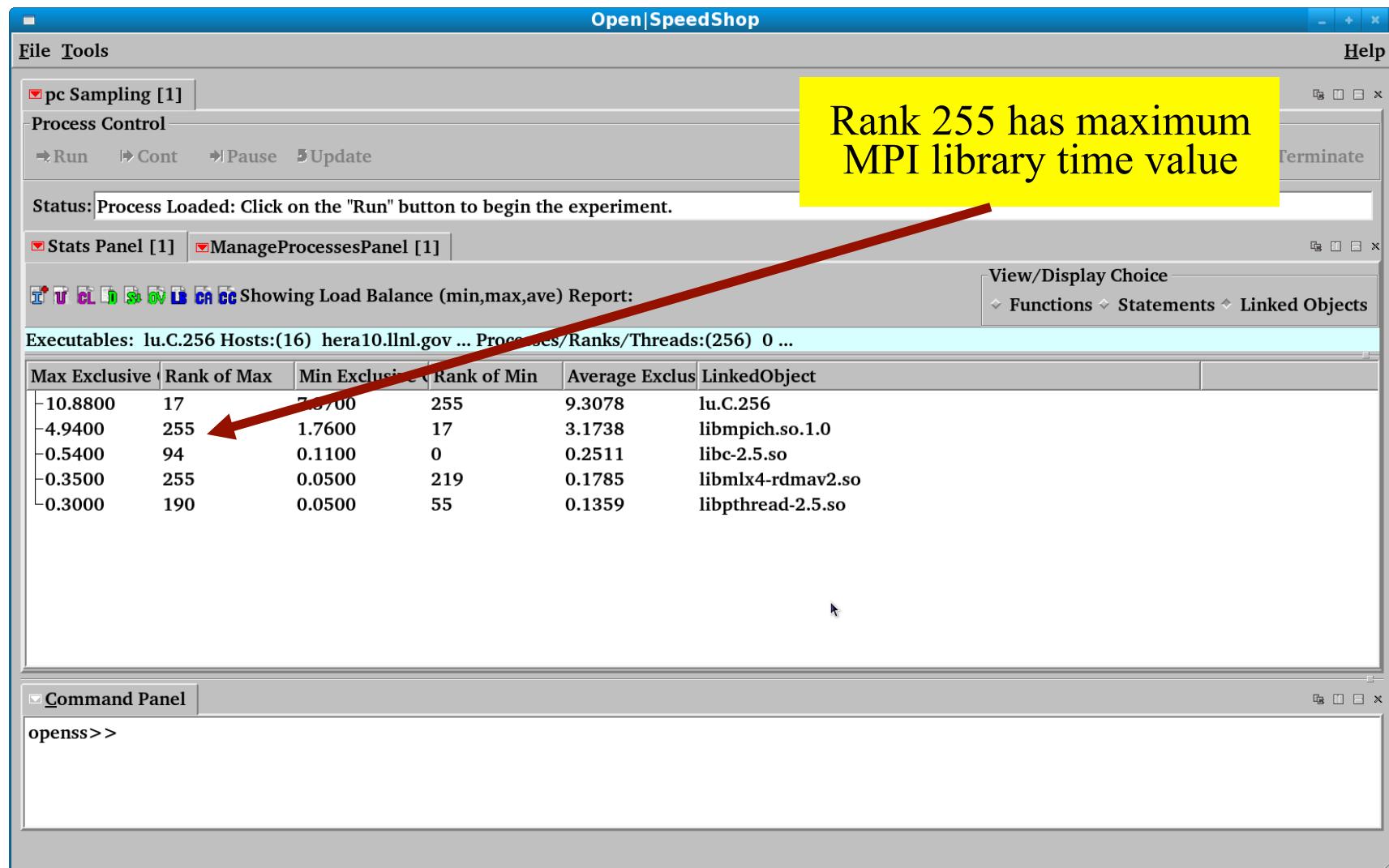
A MPI Case Study: Using NPB: LU

❖ Default View based on Linked Objects (libraries)



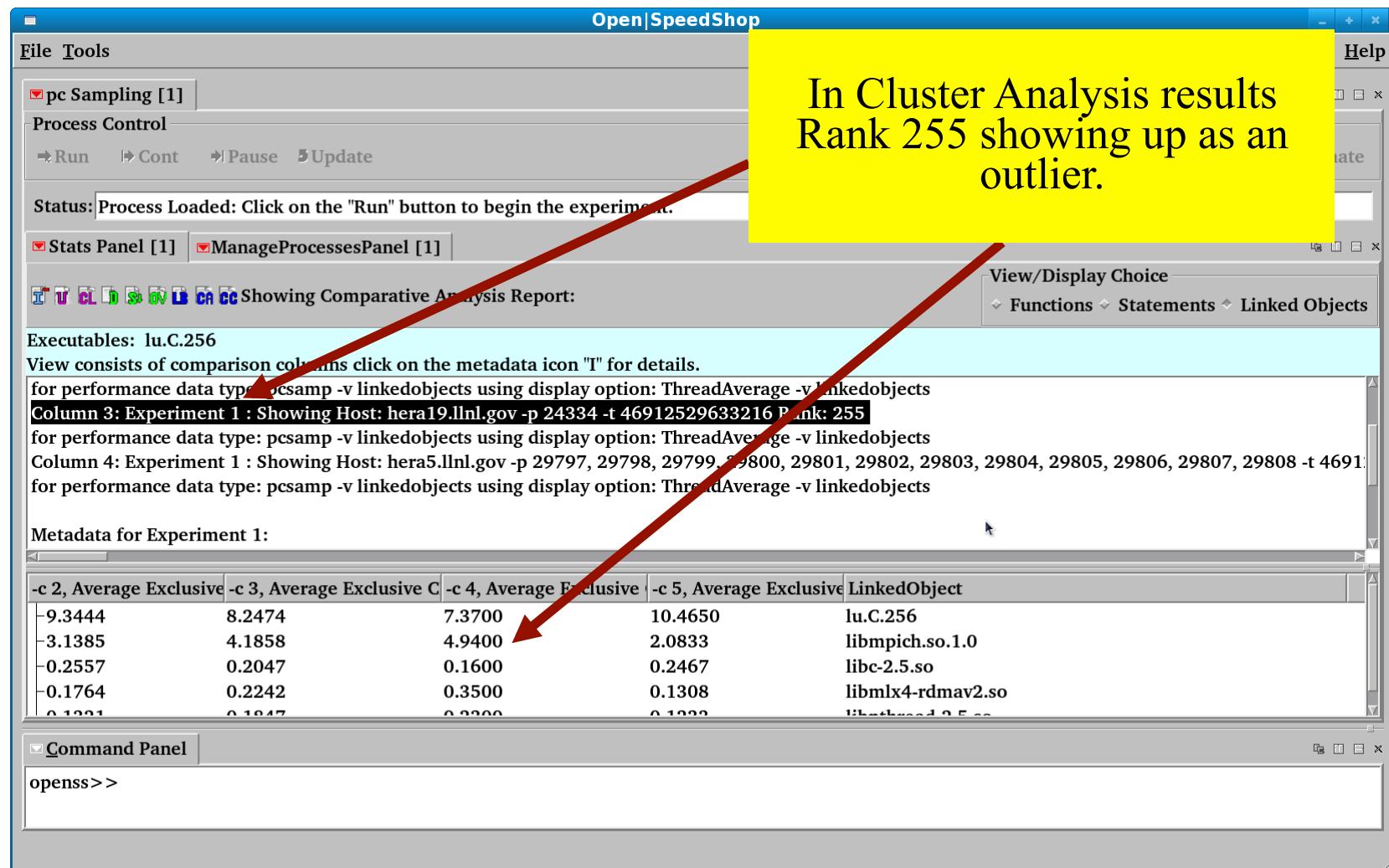
A MPI Case Study: Using NPB: LU

❖ Load Balance View based on Linked Objects (libraries)



A MPI Case Study: Using NPB: LU

❖ Cluster Analysis View based on Linked Objects (libraries)



In Cluster Analysis results
Rank 255 showing up as an outlier.

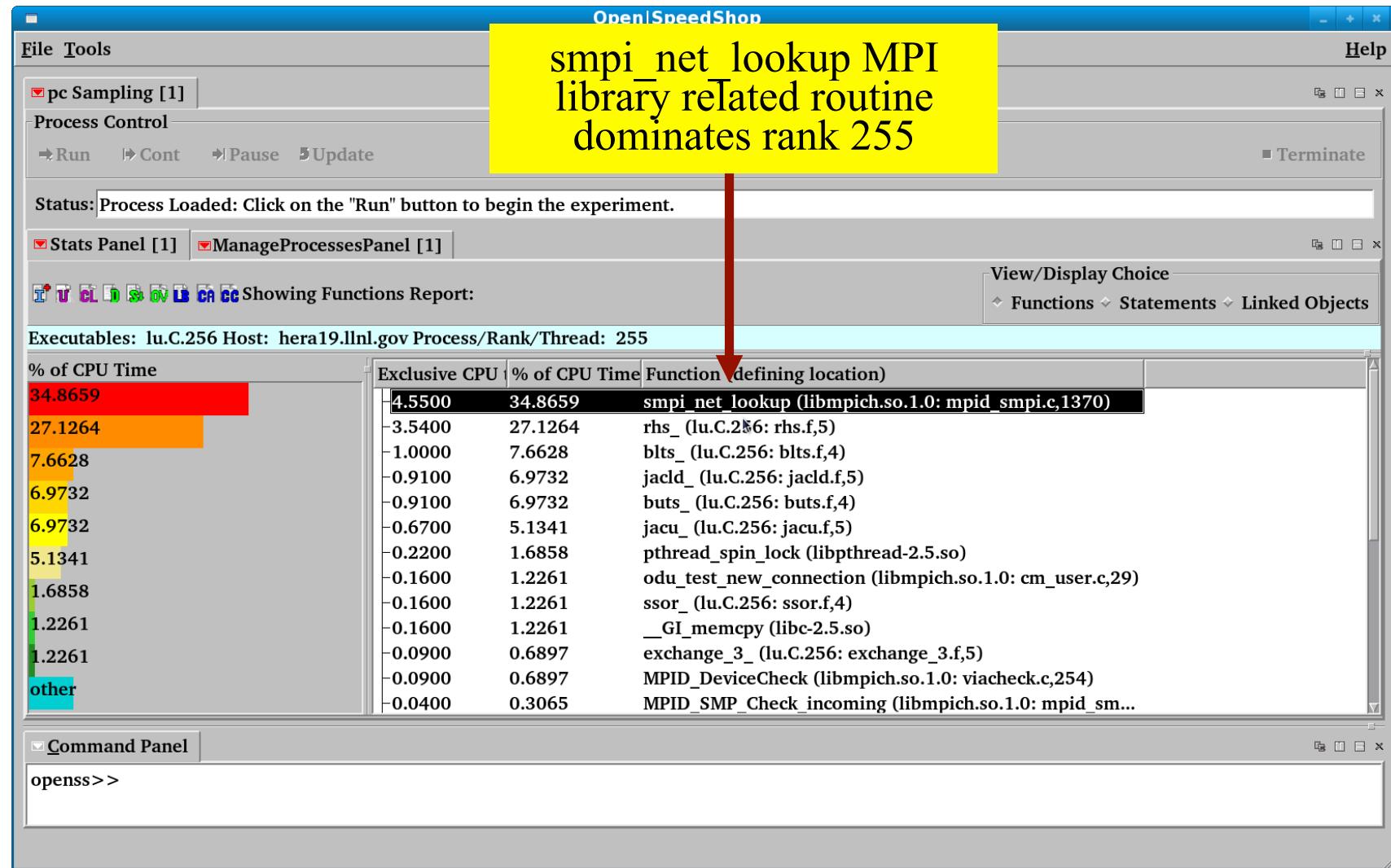
for performance data type: pcsamp -v linkedobjects using display option: ThreadAverage -v linkedobjects
Column 3: Experiment 1 : Showing Host: hera19.llnl.gov -p 24334 -t 46912529633216 Rank: 255
for performance data type: pcsamp -v linkedobjects using display option: ThreadAverage -v linkedobjects
Column 4: Experiment 1 : Showing Host: hera5.llnl.gov -p 29797, 29798, 29799, 29800, 29801, 29802, 29803, 29804, 29805, 29806, 29807, 29808 -t 4691
for performance data type: pcsamp -v linkedobjects using display option: ThreadAverage -v linkedobjects

Metadata for Experiment 1:

| -c 2, Average Exclusive | -c 3, Average Exclusive | C | -c 4, Average Exclusive | -c 5, Average Exclusive | LinkedObject |
|-------------------------|-------------------------|--------|-------------------------|-------------------------|-------------------|
| -9.3444 | 8.2474 | 7.3700 | 10.4650 | 10.4650 | lu.C.256 |
| -3.1385 | 4.1858 | 4.9400 | 2.0833 | 2.0833 | libmpich.so.1.0 |
| -0.2557 | 0.2047 | 0.1600 | 0.2467 | 0.2467 | libc-2.5.so |
| -0.1764 | 0.2242 | 0.3500 | 0.1308 | 0.1308 | libmlx4-rdmav2.so |
| 0.1001 | 0.1001 | 0.0000 | 0.1000 | 0.1000 | |

openss>>

❖ Pcsamp View of Rank 255 Performance Data only



A MPI Case Study: Using NPB: LU

❖ MPI Experiment Load Balance View (CLI)

openss>>expview -m loadbalance

| Max MPI Call Time (defining location) | Rank of Max | Min MPI Call Time | Rank of Min | Average MPI Call Function |
|--|-------------|-------------------|-------------|---|
| Across Ranks(ms) | | Across Ranks(ms) | | Time Across Ranks(ms) |
| 150332.97 | 0 | 120351.97 | 36 | 131361.13 MPI_Recv (libmpich.so.1.0: recv.c,60) |
| 17636.11 | 36 | 1103.53 | 0 | 5443.08 MPI_Send (libmpich.so.1.0: send.c,65) |
| 16470.53 | 19 | 353.81 | 0 | 5255.33 MPI_Wait (libmpich.so.1.0: wait.c,51) |
| 3206.45 | 255 | 3.00 | 17 | 2000.27 MPI_Allreduce (libmpich.so.1.0: allreduce.c,59) |
| 915.17 | 54 | 754.39 | 83 | 792.07 PMPI_Init (libmonitor.so.0.0.0: pmpi.c,94) |
| 16.00 | 48 | 5.63 | 249 | 7.29 PMPI_Finalize (libmonitor.so.0.0.0: pmpi.c,223) |
| 9.28 | 230 | 2.59 | 0 | 7.55 MPI_Irecv (libmpich.so.1.0: irecv.c,48) |
| 1.22 | 247 | 0.07 | 0 | 1.10 MPI_Bcast (libmpich.so.1.0: bcast.c,81) |
| 0.51 | 53 | 0.35 | 239 | 0.41 MPI_Barrier (libmpich.so.1.0: barrier.c,56) |

openss>>

MPI Experiment shows
Rank 255 spending significant time
in MPI_Allreduce

A MPI Case Study: Using NPB: LU

❖ MPI Rank 255 Results

```
openss>>expview -r 255 -m exclusive_time
```

Exclusive MPI Call Function (defining location)

| Time(ms) |
|--|
| 138790.370000 MPI_Recv (libmpich.so.1.0: recv.c,60) |
| 8841.088000 MPI_Wait (libmpich.so.1.0: wait.c,51) |
| 3337.737000 MPI_Send (libmpich.so.1.0: send.c,65) |
| 3206.454000 MPI_Allreduce (libmpich.so.1.0: allreduce.c, 59) |
| 797.964000 PMPI_Init (libmonitor.so.0.0.0: pmpi.c,94) |
| 5.887000 PMPI_Finalize (libmonitor.so.0.0.0: pmpi.c,223) |
| 4.701000 MPI_Irecv (libmpich.so.1.0: irecv.c,48) |
| 1.221000 MPI_Bcast (libmpich.so.1.0: bcast.c,81) |
| 0.396000 MPI_Barrier (libmpich.so.1.0: barrier.c,56) |

MPI Rank 0 Results

```
openss>>expview -r 0 -m exclusive_time
```

Exclusive MPI Call Function (defining location)

| Time(ms) |
|--|
| 150332.974000 MPI_Recv (libmpich.so.1.0: recv.c,60) |
| 1103.539000 MPI_Send (libmpich.so.1.0: send.c,65) |
| 807.433000 PMPI_Init (libmonitor.so.0.0.0: pmpi.c,94) |
| 353.810000 MPI_Wait (libmpich.so.1.0: wait.c,51) |
| 15.643000 PMPI_Finalize (libmonitor.so.0.0.0: pmpi.c, 223) |
| 8.903000 MPI_Allreduce (libmpich.so.1.0: allreduce.c,59) |
| 2.995000 MPI_Irecv (libmpich.so.1.0: irecv.c,48) |
| 0.438000 MPI_Barrier (libmpich.so.1.0: barrier.c,56) |
| 0.076000 MPI_Bcast (libmpich.so.1.0: bcast.c,81) |

MPI Experiment comparison of rank 255 to another (rank 0) shows
Rank 255 spending much more time in
MPI_Allreduce and MPI_Wait

A MPI Case Study: Using NPB: LU

❖ Hot Call Paths for MPI_Wait for rank 255 only

openss>>expview -r 255 -vcalltrees,fullstack -f MPI_Wait

| Exclusive MPI Call Time(ms) | % of Total | Number of Calls Call Stack Function (defining location) |
|-----------------------------|------------|--|
| 6010.978000 | 3.878405 | >>>main (lu.C.256) >>>> @ 140 in MAIN__ (lu.C.256: lu.f,46) >>>>> @ 180 in ssor_ (lu.C.256: ssor.f,4) >>>>>> @ 213 in rhs_ (lu.C.256: rhs.f,5) >>>>>>> @ 224 in exchange_3_ (lu.C.256: exchange_3.f,5) >>>>>>>> @ 893 in mpi_wait_ (mpi-mvapich-rt-offline.so: wrappers-fortran.c,893) >>>>>>>>> @ 889 in mpi_wait (mpi-mvapich-rt-offline.so: wrappers-fortran.c,885) 250 >>>>>>>>>> @ 51 in MPI_Wait (libmpich.so.1.0: wait.c,51) |
| 2798.770000 | 1.805823 | >>>main (lu.C.256) >>>> @ 140 in MAIN__ (lu.C.256: lu.f,46) >>>>> @ 180 in ssor_ (lu.C.256: ssor.f,4) >>>>>> @ 64 in rhs_ (lu.C.256: rhs.f,5) >>>>>>> @ 88 in exchange_3_ (lu.C.256: exchange_3.f,5) >>>>>>>> @ 893 in mpi_wait_ (mpi-mvapich-rt-offline.so: wrappers-fortran.c,893) >>>>>>>>> @ 889 in mpi_wait (mpi-mvapich-rt-offline.so: wrappers-fortran.c,885) 250 >>>>>>>>>> @ 51 in MPI_Wait (libmpich.so.1.0: wait.c,51) |

Most expensive call path to MPI_Wait

❖ Summary of study on LU

- Did a program counter sampling experiment to get overview
 - Noticed that smp_net_lookup showing up in function load balance
 - Caused us to take a look at the linked object view
 - Load balance on linked objects showed some imbalance
 - Did a cluster analysis view and found that rank 255 is an outlier
- Take a closer look at rank 255
 - Saw that rank 255's pcsamp output shows most time in smp_net_lookup
 - Run an MPI experiment to determine if we can get more clues
 - Saw that a load balance view on the MPI experiment shows rank 255's MPI_Allreduce time is highest of the 256 ranks used
 - Show rank 255 and a representative rank from rest of ranks
 - Note the differences in MPI_Wait, MPI_Send, and MPI_Allreduce
 - Look at Call Paths to MPI_Wait
- Conclusions?
 - We've found the call paths to examine for why the wait is occurring.

❖ We can do similar techniques for threaded programs

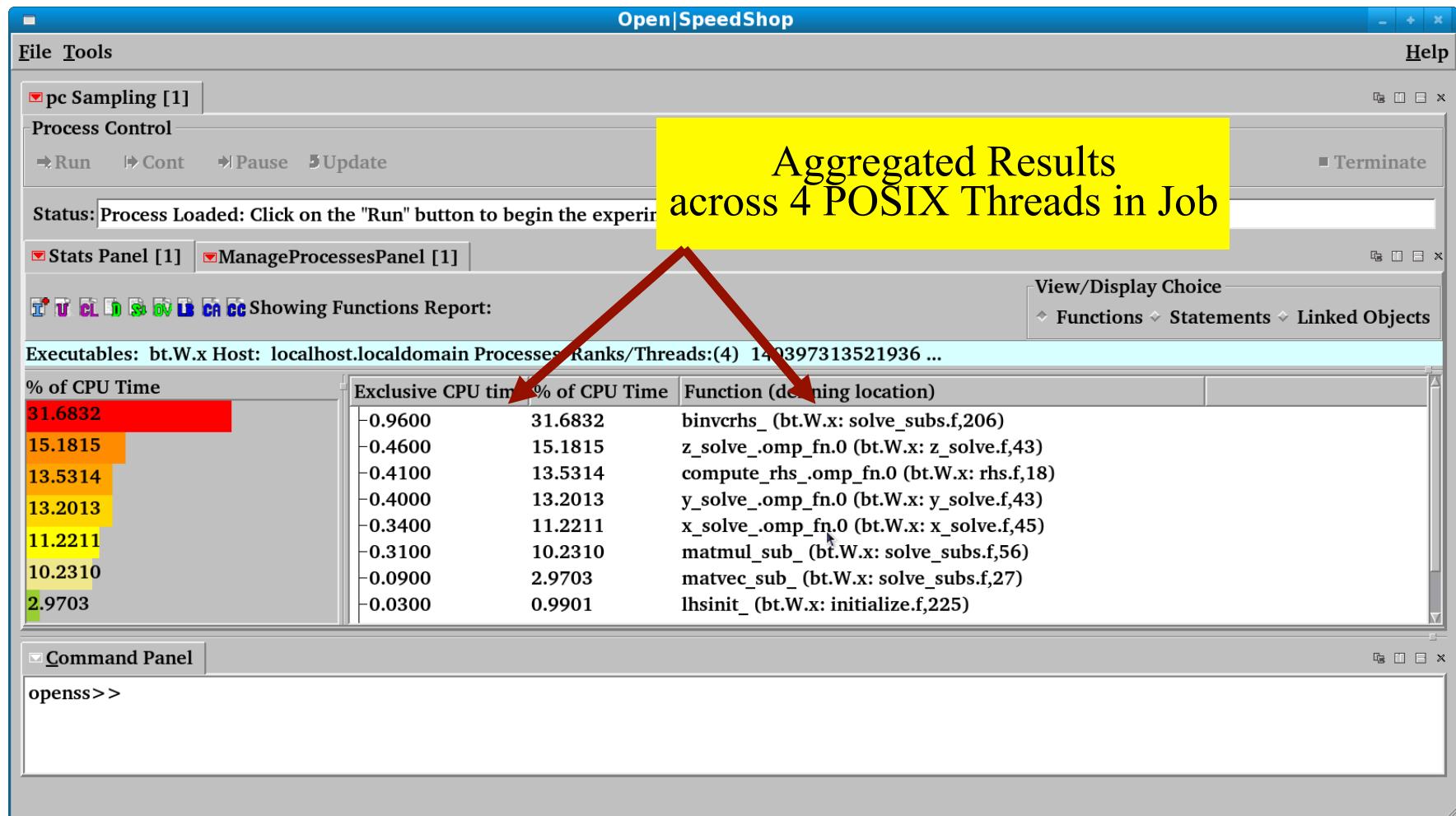
- Show first steps to analyzing the BT kernel in the OMP NPB
 - Run pcsamp experiment to get overview
 - Look at load balance view to detect if any widely varying values
 - Do cluster analysis to find outliers
- Because of time constraints the remainder of the analysis is left to the reader.

But it would include:

- Run usertime experiment to get call path information
- Examine call paths to determine if one or more paths stands out
- Examine individual thread results

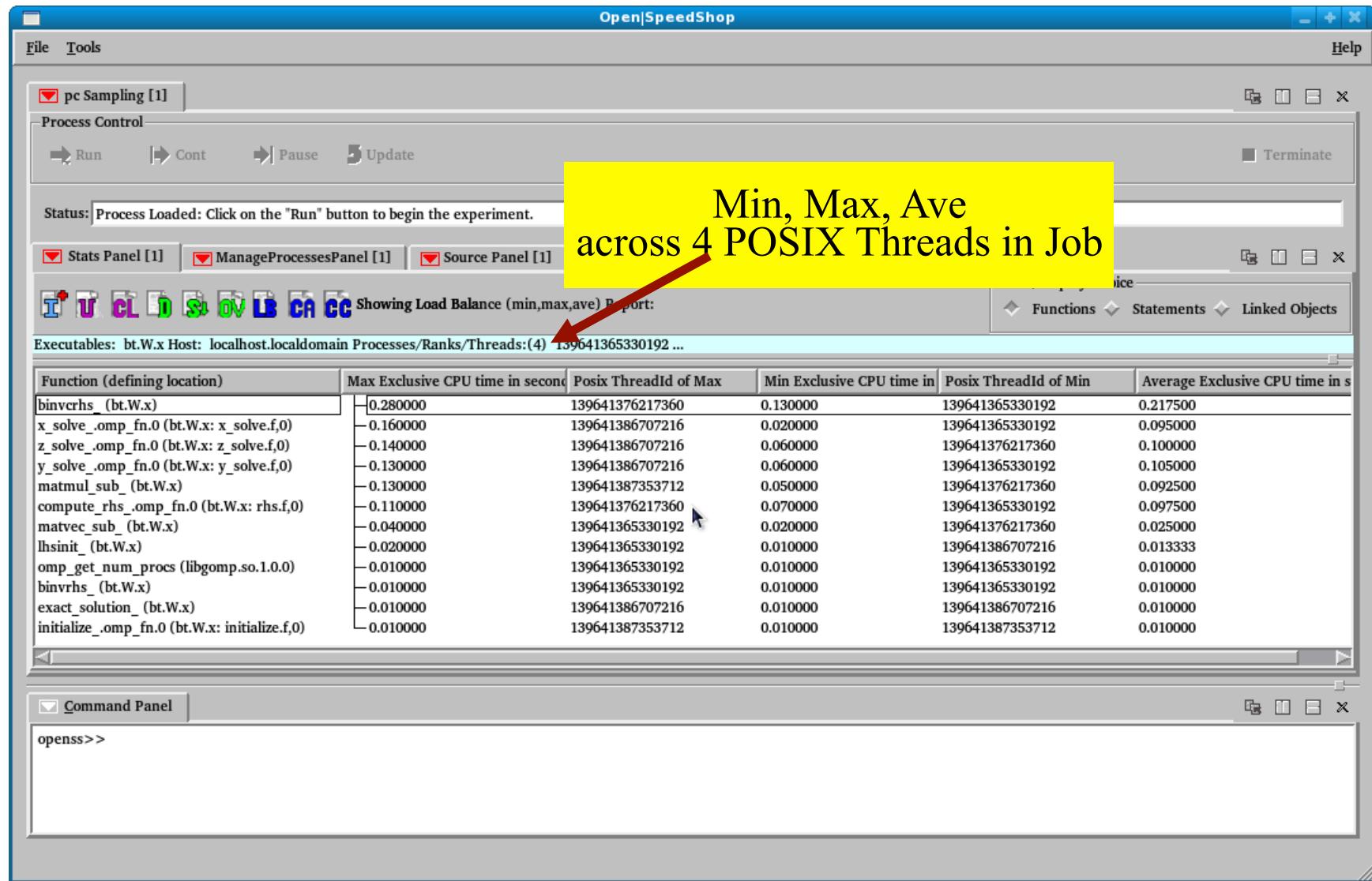
A Threads Case Study Using NPB: BT

❖ Default View based on Functions for BT (OMP_NUM_THREADS=4)



A Threads Case Study Using NPB: BT

❖ Load Balance View based on Functions



A Threads Case Study Using NPB: BT

❖ Cluster Analysis View based on Functions

Open|SpeedShop

File Tools

pc Sampling [1] Command Panel

Process Control

Run Cont Pause Update Terminate

Status: Process Loaded: Click on the "Run" button to begin the experiment.

Stats Panel [1] ManageProcessesPanel [1]

Comparative Analysis Report: Showing Comparative Analysis Report.

choice

Functions Statements Linked Objects

Executables: bt.W.x

View consists of comparison columns click on the metadata icon "I" for details.

Comparing:

Column 1: Experiment 1 : Showing Host: localhost.localdomain -p 20375 -t 140397313521936, 140397531248912, 140397541738768 Rank: -1
for performance data type: pcsamp -v functions using display option: ThreadAverage -v functions

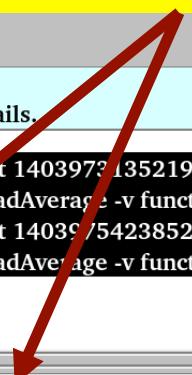
Column 2: Experiment 1 : Showing Host: localhost.localdomain -p 20375 -t 140397542385264 Rank: -1
for performance data type: pcsamp -v functions using display option: ThreadAverage -v functions

Metadata for Experiment 1:

-c 2, Average Exclusive CPU time in seconds. Across Threads | -c 3, Average Exclusive CPU time in seconds | Function (defining location)

| -0.1600 | 0.4800 | binverhs_ (bt.W.x: solve_subs.f,206) |
|---------|--------|---|
| -0.1033 | 0.1500 | z_solve_omp_fn.0 (bt.W.x: z_solve.f,43) |
| -0.0900 | 0.1400 | compute_rhs_omp_fn.0 (bt.W.x: rhs.f,18) |
| -0.0900 | 0.1300 | y_solve_omp_fn.0 (bt.W.x: y_solve.f,43) |
| -0.0633 | 0.1200 | matmul_sub_ (bt.W.x: solve_subs.f,56) |

Cluster Analysis across 4 POSIX Threads in Job Shows one outlier



- ❖ **mpit** experiment has a performance information entry for each MPI function call.
- ❖ In addition to the time spent in each MPI function, information like source and destination rank, bytes sent or received are also available.
 - Dependent on call
- ❖ Can selectively view the information desired
- ❖ Can be bursty and no Open|SpeedShop graphical viewer for this information
- ❖ Can use **mpiotf** experiment, then use **Vampir**

Tracing Results: Event View (default)

Open|SpeedShop

File Tools Help

MPIT [1]

Process Control: Run, Cont, Pause, Update, Terminate

Status: Process Loaded: Click on the "Run" button to begin the experiment.

Stats Panel [1] ManageProcessesPanel [1]

I U CL D G GI HC B TS OV EL LB CA CC Showing Per Event Report Functions

Executables: smg2000 Host: localhost.localdomain Processes/Ranks/Threads:(2) 0 ...

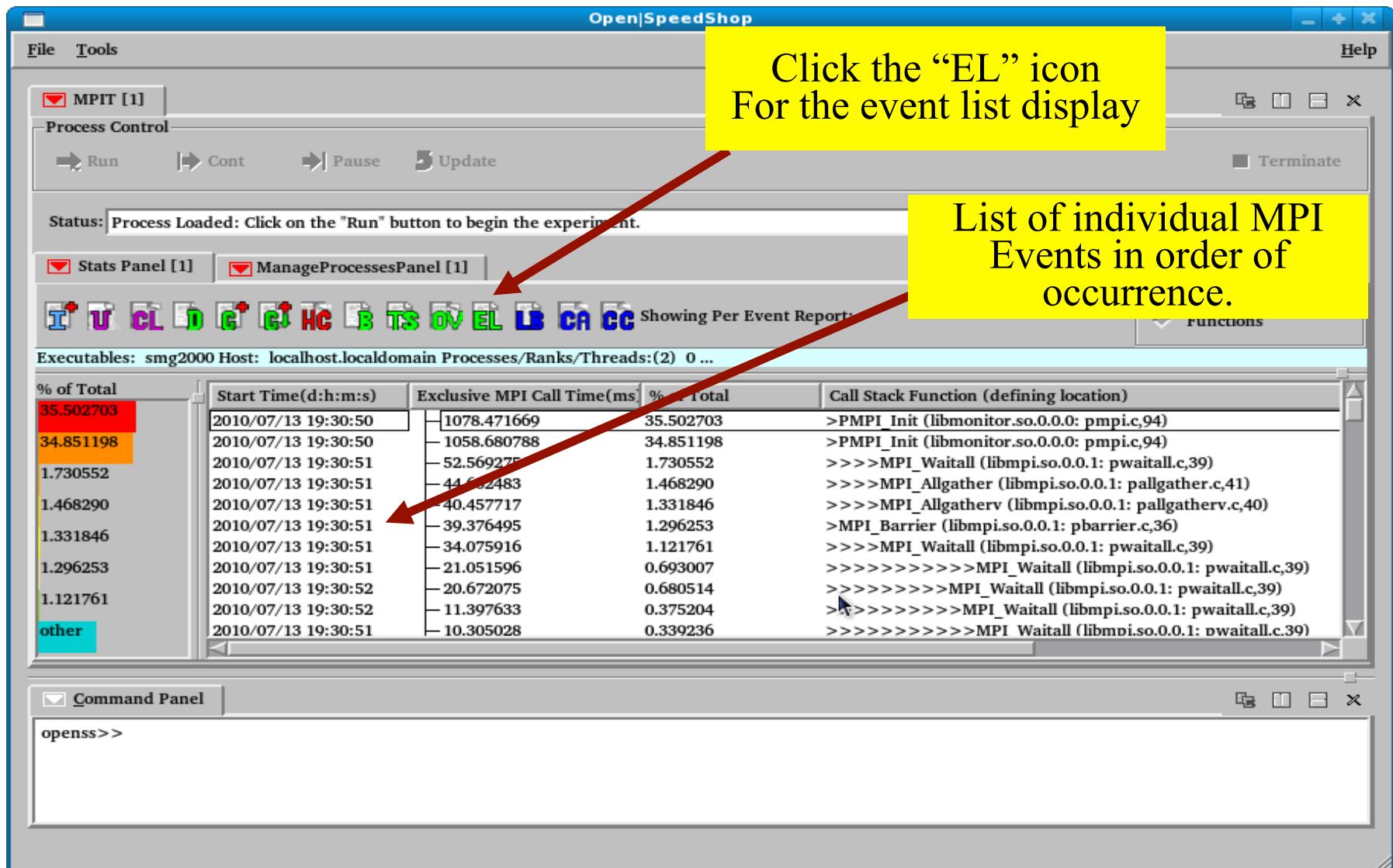
% of Total

| % of Total | Start Time(d:h:m:s) | Exclusive MPI Call Time(ms) | % of Total | Call Stack Function (defining location) |
|------------|---------------------|-----------------------------|------------|---|
| 35.502703 | 2010/07/13 19:30:50 | 1078.471669 | 35.502703 | >PMPI_Init (libmonitor.so.0.0.0: pmpi.c,94) |
| 34.851198 | 2010/07/13 19:30:50 | 1058.680788 | 34.851198 | >PMPI_Init (libmonitor.so.0.0.0: pmpi.c,94) |
| 1.730552 | 2010/07/13 19:30:51 | 52.569275 | 1.730552 | >>>MPI_Waitall (libmpi.so.0.0.1: pwaitall.c,39) |
| 1.468290 | 2010/07/13 19:30:51 | 44.62483 | 1.468290 | >>>MPI_Allgather (libmpi.so.0.0.1: pallgather.c,41) |
| 1.331846 | 2010/07/13 19:30:51 | 40.457717 | 1.331846 | >>>MPI_Allgatherv (libmpi.so.0.0.1: pallgatherv.c,40) |
| 1.296253 | 2010/07/13 19:30:51 | 39.376495 | 1.296253 | >MPI_BARRIER (libmpi.so.0.0.1: pbarrier.c,36) |
| 1.121761 | 2010/07/13 19:30:51 | 34.075916 | 1.121761 | >>>MPI_Waitall (libmpi.so.0.0.1: pwaitall.c,39) |
| other | 2010/07/13 19:30:51 | 21.051596 | 0.693007 | >>>>>>MPI_Waitall (libmpi.so.0.0.1: pwaitall.c,39) |
| | 2010/07/13 19:30:52 | 20.672075 | 0.680514 | >>>>>>MPI_Waitall (libmpi.so.0.0.1: pwaitall.c,39) |
| | 2010/07/13 19:30:52 | 11.397633 | 0.375204 | >>>>>>MPI_Waitall (libmpi.so.0.0.1: pwaitall.c,39) |
| | 2010/07/13 19:30:51 | 10.305028 | 0.339236 | >>>>>>MPI_Waitall (libmpi.so.0.0.1: pwaitall.c,39) |

Command Panel: openss>>

Click the “EL” icon
For the event list display

List of individual MPI Events in order of occurrence.



Tracing Results: Creating Event View

Open|SpeedShop

File Tools Help

MPIT [1]

Process Control

Run Cont Pause Update Terminate

Status: Process Loaded: Click on the "Run" button to begin the experiment.

Stats Panel [1] ManageProcessesPanel [1]

I U CL F G C HC B TS OV EL LB CA CC Showing Functions Report:

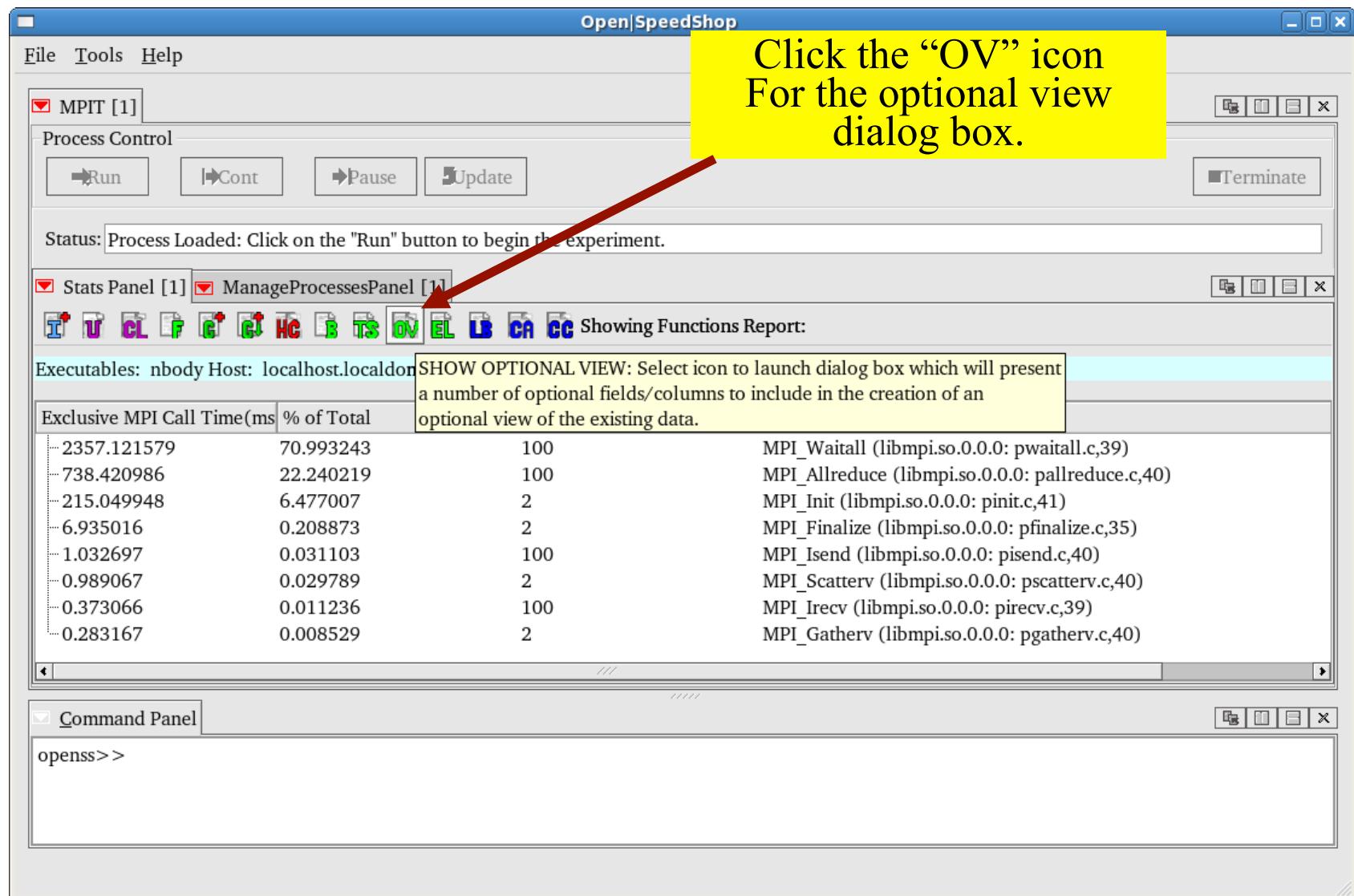
Executables: nbody Host: localhost.localdomain SHOW OPTIONAL VIEW: Select icon to launch dialog box which will present a number of optional fields/columns to include in the creation of an optional view of the existing data.

| Exclusive MPI Call Time(ms) | % of Total | | |
|-----------------------------|------------|-----|--|
| 2357.121579 | 70.993243 | 100 | MPI_Waitall (libmpi.so.0.0.0: pwaitall.c,39) |
| 738.420986 | 22.240219 | 100 | MPI_Allreduce (libmpi.so.0.0.0: pallreduce.c,40) |
| 215.049948 | 6.477007 | 2 | MPI_Init (libmpi.so.0.0.0: pinit.c,41) |
| 6.935016 | 0.208873 | 2 | MPI_Finalize (libmpi.so.0.0.0: pfinalize.c,35) |
| 1.032697 | 0.031103 | 100 | MPI_Isend (libmpi.so.0.0.0: pisend.c,40) |
| 0.989067 | 0.029789 | 2 | MPI_Scatterv (libmpi.so.0.0.0: pscatterv.c,40) |
| 0.373066 | 0.011236 | 100 | MPI_Irecv (libmpi.so.0.0.0: pirecv.c,39) |
| 0.283167 | 0.008529 | 2 | MPI_Gatherv (libmpi.so.0.0.0: pgatherv.c,40) |

Command Panel

```
openss>>
```

Click the “OV” icon
For the optional view dialog box.

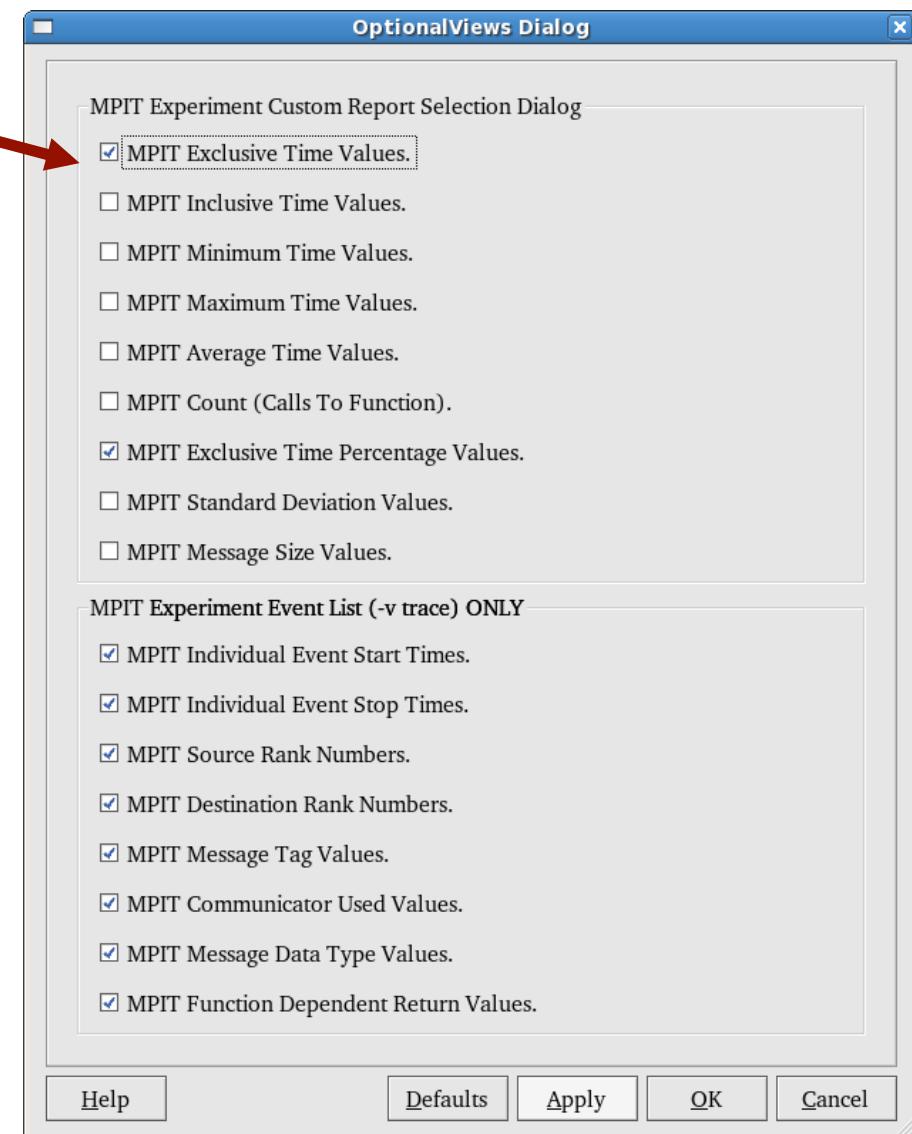


Tracing Results: Create Event View (2)

Select the metric values you want to see in the display and click OK

Use the Optional Views Dialog box to choose the performance metrics to be displayed in the StatsPanel and click OK

Clicking OK will regenerate the StatsPanel with the new metrics displayed



Tracing Results: Specialized Event View

Note:
The newly selected metrics are now being displayed.



Open|SpeedShop

File Tools Help

MPIT [1]

Process Control

Run Cont Pause Update Terminate

Status: Process Loaded: Click on the "Run" button to begin the experiment.

Stats Panel [1] ManageProcessesPanel [1]

I U CL D C H C B TS OV EL LB CA CC Showing Functions Report:

View/Display Choice

Functions

Executables: smg2000 Host: localhost.localdomain Processes/Ranks/Threads:(2)

| % of Total | Start Time(d:h:m:s) | Exclusive MPI Call T | % of Total | Source R | Dest | Message Tag | Call Stack Function (defining location) |
|------------|---------------------|----------------------|------------|----------|------|-------------|---|
| 35.502703 | 2010/07/13 19:30:50 | - 1058.680788 | 34.851198 | 32767 | | 68 | >PMPI_Init (libmonitor.so.0.0.0: pmpi.c,94) |
| 34.851198 | 2010/07/13 19:30:50 | - 1078.471669 | 35.502703 | 32767 | 1 | 68 | >PMPI_Init (libmonitor.so.0.0.0: pmpi.c,94) |
| 1.730552 | 2010/07/13 19:30:51 | - 10.305028 | 0.339236 | 1 | 1 | 1548167912 | >>>>>>>>MPI_Waitall (libmpi.so.0.0.1: pwai |
| 1.468290 | 2010/07/13 19:30:51 | - 2.615438 | 0.086099 | 2 | | 601616488 | >>>>>>>>MPI_Waitall (libmpi.so.0.0.1: pw |
| 1.331846 | 2010/07/13 19:30:51 | - 0.439376 | 0.014464 | 2 | | 601617320 | >>>>>>>>MPI_Waitall (libmpi.so.0.0.1: pwaitall. |
| 1.296253 | 2010/07/13 19:30:51 | - 0.443217 | 0.014590 | 1 | | 601617000 | >>>>>>>>MPI_Waitall (libmpi.so.0.0.1: pwaita |
| 1.121761 | 2010/07/13 19:30:51 | - 0.341387 | 0.011238 | 1 | | 601617000 | >>>>>>>>MPI_Waitall (libmpi.so.0.0.1: pwaita |
| other | 2010/07/13 19:30:51 | - 34.075916 | 1.121761 | | | 30346816 | >>>MPI_Waitall (libmpi.so.0.0.1: pwaitall.c,39) |
| | 2010/07/13 19:30:51 | - 21.051596 | 0.693007 | 2 | 1 | 1548167912 | >>>>>>>>MPI_Waitall (libmpi.so.0.0.1: pwai |
| | 2010/07/13 19:30:51 | - 0.365065 | 0.012018 | | | 1 | >>>>>>MPI_Waitall (libmpi.so.0.0.1: pwaitall.c,36 |
| | 2010/07/13 19:30:51 | - 39.376495 | 1.296253 | | | | >MPI_Barrier (libmpi.so.0.0.1: barrier.c,36) |

Command Panel

openss>>

❖ Open|SpeedShop manages MPI jobs

- Works with multiple MPI implementations
- Process control using MPIR interface (dynamic version)

❖ Parallel experiments

- Apply the sequential O|SS collectors to all nodes
- Specialized MPI tracing experiments

❖ Result Viewing

- By default: results are aggregated across ranks/threads
- Optionally: select individual ranks/threads or groups
- Compare or group ranks
- Specialized views:
 - Load balance
 - Comparative analysis / cluster analysis

❖ Use features to isolate sections of problem code

Open | SpeedShop™

Section 7

Comparing Experiment Results

SciDAC 2011 Tutorial
How to Analyze the Performance of Parallel Codes 101

A case study with Open|SpeedShop



❖ GUI Custom Compare Panel (CC icon)

- See users guide or tutorials for details

❖ Convenience Script: osscompare

- Compares Open|SpeedShop databases to each other
- Syntax: osscompare "db1.openss,db2.openss,..." [options]
- osscompare man page has more details
- Produces side-by-side comparison listing
- Limit the number of lines by "rows=nn" option
- Compare up to 8 at one time
 - If output file name is too long use:
 - export **OPENSS_USE_INTERNAL_NAMING=1**
 - Causes shortened output file names to be created
- Optionally create "csv" output for input into spreadsheet (Excel,..)
 - export **OPENSS_CREATE_CSV=1**

Comparing Performance Data (2)

osscompare "smg2000-pcsamp.openss,smg2000-pcsamp-1.openss"

openss]: Legend: -c 2 represents smg2000-pcsamp.openss

[openss]: Legend: -c 4 represents smg2000-pcsamp-1.openss

-c 2, Exclusive CPU -c 4, Exclusive CPU Function (defining location)

time in seconds. time in seconds.

3.870000000 3.630000000 hypre_SMGResidual (smg2000: smg_residual.c,152)

2.610000000 2.860000000 hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)

2.030000000 0.150000000 opal_progress (libopen-pal.so.0.0.0)

1.330000000 0.100000000 mca_btl_sm_component_progress (libmpi.so.0.0.2:
topo_unity_component.c,0)

0.280000000 0.210000000 hypre_SemiInterp (smg2000: semi_interp.c,126)

0.280000000 0.040000000 mca_pml_ob1_progress (libmpi.so.0.0.2: topo_unity_component.c,
0)

Open | SpeedShop™

Section 8 User Interfaces

SciDAC 2011 Tutorial
How to Analyze the Performance of Parallel Codes 101
A case study with Open/SpeedShop



❖ GUI panel management

- Peel-off and rearrange any panel
- Color coded panel groups pre experiment

❖ Context sensitive menus

- Right click at any location
- Access to different views
- Activate additional panels

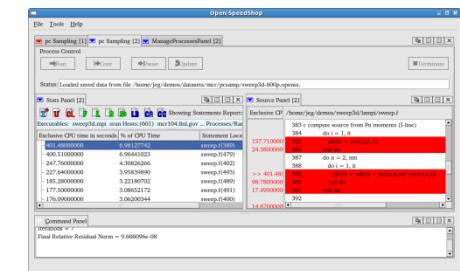
❖ Access to source location of events

- Double click on stats panel
- Opens source panel with (optional) statistics

❖ Examples throughout presentation

❖ Three different option to run without the GUI

- Equal functionality
- Can transfer state/results



❖ Interactive Command Line Interface

```
[1052]{dmont@tu-fe1:}$ openss -cli  
Welcome to OpenSpeedShop 2.0.1  
openss>>
```

❖ Immediate Command (Batch) Interface

- `openss -batch < openss_cmd_file`
- `openss -batch -f <exe> <experiment>`

❖ Python Scripting API

- `python openss_python_script_file.py`

❖ An interactive Command Line Interface

- gdb/dbx like processing

❖ Several interactive commands

- Create Experiments
- Provide Process/Thread Control (online version)
- View Experiment Results

❖ Where possible, commands execute asynchronously

http://www.openspeedshop.org/docs/cli_doc/

❖ Experiment creations

- expcreate
- expattach*

❖ Experiment control

- expgo
- expwait*
- expdisable*
- expenable*

❖ Experiment storage

- expsave
- exprestore

❖ Result presentation

- expview
- opengui

❖ Misc. commands

- help
- list
- log
- record
- playback
- history
- quit

*online version only

❖ Simple usage to create, run, view data

- **openss –cli** (use cli to run experiment:3 commands)
 - **expcreate –f “mutatee 2000” pcsamp** (create an experiment with instrumentation added for the particular collector)
 - **expgo** (runs the experiment gathering data into database)
 - **expview** (displays the default view of the performance data)

❖ Alternative views of the performance data

- **expview –v statements** (see the statements that took the most time)
- **expview –v linkedobjects** (see time attributed to the libraries in appl.)
- **expview –v calltrees, fullstack** (see all the call paths in application)
- **expview –m loadbalance** (see the min, max, average across ranks/ threads)
- **list –v metrics** (display the optional performance data metrics)
- **expview –m <metric from above>** (view the metric specified)

❖ Want the GUI while in CLI?

- **opengui** – raises the GUI and assumes the experiment info

User-Time Example (CLI run experiment)

Create experiments
and load application
named “fred”

```
lnx17>openss -cli  
openss>>Welcome to openSpeedShop 2.0.1  
openss>>expcreate -f test/executables/fred/fred usertime
```

The new focused experiment identifier is: -x 1

```
openss>>expgo
```

Start application

Start asynchronous execution of experiment: -x 1

```
openss>>Experiment 1 has terminated.
```

Showing CLI Results

```
openss>>expview
```

Show default view

| Function | Excl CPU time in seconds. | Incl CPU time in seconds. | % of Total | Exclusive CPU Time (defining location) |
|----------|------------------------------|------------------------------|------------|--|
| 5.2571 | 5.2571 | 5.2571 | 49.7297 | f3 (fred: f3.c, 2) |
| 3.3429 | 3.3429 | 3.3429 | 31.6216 | f2 (fred: f2.c, 2) |
| 1.9714 | 1.9714 | 1.9714 | 18.6486 | f1 (fred: f1.c, 2) |
| 0.0000 | 10.5429 | 10.5429 | 0.0000 | work(fred:work.c, 2) |
| 0.0000 | 10.5714 | 10.5714 | 0.0000 | main (fred: fred.c, 5) |

❖ Create batch file with CLI commands

- Plain text file
- Example:

```
# Create batch file
echo expcreate -f fred pcsamp >> input.script
echo expgo >> input.script
echo expview pcsamp10 >>input.script

# Run OpenSpeedShop
openss -batch < input.script
```

❖ Open|SpeedShop batch example results

```
The new focused experiment identifier is: -x 1  
Start asynchronous execution of experiment: -x 1
```

```
Experiment 1 has terminated.
```

| CPU Time | Function (defining location) |
|----------|-------------------------------|
| 24.2700 | f3 (mutatee: mutatee.c, 24) |
| 16.0000 | f2 (mutatee: mutatee.c, 15) |
| 8.9400 | f1 (mutatee: mutatee.c, 6) |
| 0.0200 | work (mutatee: mutatee.c, 33) |

- ❖ Note: Currently this interface is only supported via the online version of Open|SpeedShop
 - Requires Open|SpeedShop built with OPENSS_INSTRUMENTOR=mrnet, the default is offline

Python Scripting

- ❖ Open|SpeedShop Python API that executes “same” Interactive/Batch Open|SpeedShop commands
- ❖ User can intersperse “normal” Python code with Open|SpeedShop Python API
- ❖ Run Open|SpeedShop experiments via the Open| SpeedShop Python API
- ❖ Note: Currently this interface is only supported via the online version of Open|SpeedShop
 - Requires Open|SpeedShop built with OPENSS_INSTRUMENTOR=mrnet, the default is offline

MPI_Pcontrol Support in O|SS

This feature allows the user to gather data for sections of their code bounded by MPI_Pcontrol calls inserted into the user application. Examples of the execution using this feature need to follow this pattern:

- ❖ 1) Modification of the application code to insert MPI_Pcontrol (argument) calls.
 - MPI_Pcontrol(1) enables gathering of performance data.
 - MPI_Pcontrol(0) disables gathering of performance data.
- ❖ 2) Set environment variable: OPENSS_ENABLE_MPI_PCONTROL=1 to activate the MPI_Pcontrol call recognition, otherwise the MPI_Pcontrol calls will be ignored
- ❖ 3) Optionally set: OPENSS_START_ENABLED=1 to have performance data gathered until a MPI_Pcontrol (0) call is encountered. Otherwise, no performance data will be gathered until a MPI_Pcontrol (1) call is encountered. NOTE: OPENSS_ENABLE_MPI_PCONTROL must be set for this environment variable to have any effect.

❖ Multiple non-graphical interfaces

- Interactive Command Line
- Batch scripting
- Python scripting

❖ Equal functionality

- Similar commands in all interfaces
- Same internal routines process all interface requests

❖ Results transferable

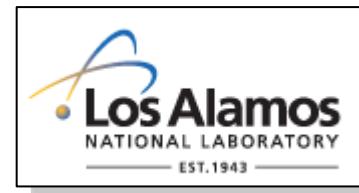
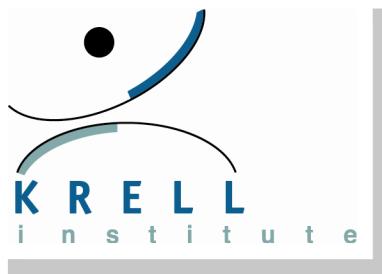
- E.g., run in Python and view in GUI
- Possibility to switch GUI ↔ CLI

❖ Allow user to control what portions of their application to gather data for using MPI_Pcontrol

Open | SpeedShop™

Section 9 Static Binary Support

SciDAC 2011 Tutorial
How to Analyze the Performance of Parallel Codes 101
A case study with Open/SpeedShop



- ❖ **When shared library support is limited**
 - Normal manner of running experiments doesn't work
 - Need to link our collectors into the static executable
- ❖ **Current method (static) of support on Cray and BG/P**
- ❖ **Must relink application to add in OSS collector code.**
 - `osslink` command is provided to help
- ❖ **Have the Open|SpeedShop target NOT host environment module loaded when relinking**

❖ osslink: A script to help with linking in our collectors

- **osslink** is a script that hides a lot of the link details
- Calls to it are usually embedded inside application makefiles
- Can also be used to compile and link applications
- Sorts the experiment specific library and collector specification
- Sorts out some platform differences to do the correct link

❖ Find Makefile target to add entry for relink with collector

- The user generally needs find the target that creates the actual static executable and create a collector target that links in the selected collector as shown in the example.

❖ Example follows ->

Re-linking application using osslink

❖ Example modification for smg2000 application

```
smg2000: smg2000.o
@echo "Linking" $@ "... "
${CC} -o smg2000 smg2000.o ${LFLAGS}
```

```
smg2000-pcsamp: smg2000.o
@echo "Linking" $@ "... "
osslink -v -c pcsamp ${CC} -o smg2000-pcsamp smg2000.o ${LFLAGS}
```

```
smg2000-usertime: smg2000.o
@echo "Linking" $@ "... "
osslink -v -c usertime ${CC} -o smg2000-usertime smg2000.o ${LFLAGS}
```

```
smg2000-hwcsamp: smg2000.o
@echo "Linking" $@ "... "
osslink -v -c hwcsamp ${CC} -o smg2000-hwcsamp smg2000.o ${LFLAGS}
```

```
smg2000-io: smg2000.o
@echo "Linking" $@ "... "
osslink -u open -v -c io ${CC} -o smg2000-io smg2000.o ${LFLAGS}
```

```
smg2000-iot: smg2000.o
@echo "Linking" $@ "... "
osslink -u open -v -c iot ${CC} -o smg2000-iot smg2000.o ${LFLAGS}
```

```
smg2000-mpi: smg2000.o
@echo "Linking" $@ "... "
osslink -v -c mpi ${CC} -o smg2000-mpi smg2000.o ${LFLAGS}
```

- ❖ Running the re-linked executable causes the application to write the raw data files to the location specified by **OPENSS_RAWDATA_DIR**
- ❖ Converting raw data to an OpenSpeedShop database
 - Normally the convenience scripts or **openss** tool will do this under the hood
 - **ossutil** command will create the database file.
 - If raw data files are in /home/jgalaro-smg2000/test/raw
 - Then **ossutil** /home/jgalaro-smg2000/test/raw # creates a database file
 - Symbol table information is read and mapped to the raw data application addresses to map source to the performance data.
- ❖ The **ossutil** step may be added to the batch script to eliminate an additional step
- ❖ Now normal Open|SpeedShop GUI and CLI viewing of the performance data is possible

Workflow Differences for Static Binary

❖ Now run executable: (note **-bb** argument to aprun)

```
PBS -q debug
#PBS -N smg2000-pcsamp
...
# must have a clean raw data directory each run
rm -rf /home/jgalaro/smg2000/test/raw
mkdir /home/jgalaro/smg2000/test/raw

setenv OPENSS_RAWDATA_DIR /home/jgalaro/smg2000/test/raw
setenv OPENSS_DB_DIR /home/jgalaro/smg2000/test/

cd /home/jgalaro/smg2000/test
# needs -bb to have the original executable available when doing ossutil
aprun -bb -n 16 /home/jgalaro/smg2000/test/smg2000-pcsamp

# creates a X.0.openss database file, may need to load module pointing to openspeedshop here
ossutil /home/jgalaro/smg2000/test/raw
```

❖ Changes to shared library support recently

- Dynamic shared library support now available in newer Cray and Blue Gene O/S
- We provide shared support in addition to static binary support on the Cray, but not yet on Blue Gene (working on it).
- This allows the normal Open|SpeedShop workflow to be used, for example:
 - osspcsamp “how you run the executable dynamically”

❖ Also, coming is a replacement mechanism for having to relink the static binaries to insert the Open|SpeedShop collectors into the application.

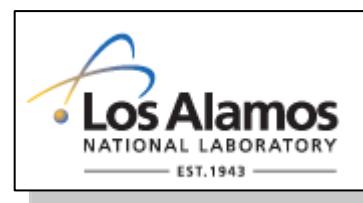
- Use Dyninst binary rewriter to insert collectors under the hood
- Then use same: osspcsamp, etc. interface for all types of applications.

Open | SpeedShop™

Section 10 How Can I Repeat this (and more) at Home?

SciDAC 2011 Tutorial
How to Analyze the Performance of Parallel Codes 101

A case study with Open|SpeedShop



❖ System architecture

- AMD Opteron/Athlon
- Intel x86, x86-64, and Itanium-2

❖ Operating system

- Tested on Many Popular Linux Distributions
 - SLES, SUSE
 - RHEL
 - Fedora Core, CentOS
 - Debian, Ubuntu
 - Varieties of the above

❖ Large scale platforms

- IBM Blue Gene/P
- Cray XT line
- For special build & usage instructions, see web site

❖ Sourceforge Project Home

- <http://sourceforge.net/projects/openss>

❖ CVS Access

- http://sourceforge.net/scm/?type=cvs&group_id=176777

❖ Packages

- Accessible from Project Home Download Tab

❖ Additional Information

- <http://www.openspeedshop.org/>

❖ install.sh

- bash script used to build the components that Open|SpeedShop uses as well as the Open|SpeedShop core
- First will check to see if you have the correct supporting software installed on your system.
 - If not, will stop and ask you if you want to continue
 - Will tell you what build variables are needed

➤ Example

```
export OPENSS_PREFIX=/home/skg/local
export OPENSS_INSTRUMENTOR=mrnet
export OPENSS_MPI_OPENMPI=/opt/openmpi-1.3.2
export OPENSS_MPI_MVAPICH=/opt/mvapich-1.1
export QTDIR=/usr/lib64/qt-3.3 (usually set up by the system login files)
```

- Will, optionally, then build and install all the prerequisite packages and also Open|SpeedShop itself.
 - Can build and install one component at a time.
 - The whole package – option 9
- Builds and installs single or groups of components so that the next components use the previous components.

❖ Important runtime environment variables

- OPENSS_PREFIX
 - Install directory path
- OPENSS_PLUGIN_PATH
 - Path to directory where plugins are stored
- OPENSS_MPI_IMPLEMENTATION (if multiple)
 - If Open|SpeedShop was built with multiple MPI implementations, this points openss at the one you are using in your application
 - Also, only required if using the mpi, mpit, or mpiof experiments
- LD_LIBRARY_PATH, PATH
 - Linux path variables
- Example

```
export OPENSS_PREFIX=/home/skg/local
export OPENSS_MPI_IMPLEMENTATION=openmpi
export OPENSS_PLUGIN_PATH=$OPENSS_PREFIX/lib64/openspeedshop
export LD_LIBRARY_PATH=$OPENSS_PREFIX/lib64:$LD_LIBRARY_PATH
export PATH=$OPENSS_PREFIX/bin:$PATH
```

❖ Do an initial O|SS build

- Start with offline
- Use GUI to run a serial application (compile with -g)
- Try various sampling experiments
- Follow documentation to understand GUI features

❖ Build O|SS with MPI path variable

- Use GUI to run parallel application and investigate results
- Try various sampling and other experiments
- Try running the same with CLI commands

❖ Build O|SS with online option

- Review documentation on MRNet topology options
- Try attaching to running application

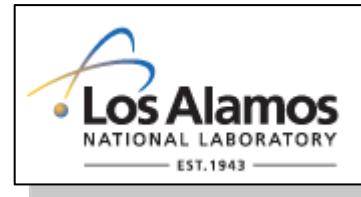
Open | SpeedShop™

Section 11

Other Projects Related to Open | SpeedShop

SciDAC 2011 Tutorial
How to Analyze the Performance of Parallel Codes 101

A case study with Open | SpeedShop



❖ Component Based Tool Framework (CBTF) project

- Provides the ability to:
 - Create “black box” components
 - Connect these components into a component network
 - Distribute these components and/or component networks across a tree base transport mechanism (MRNet).
 - Want these components to be generic enough to be reused
 - Aimed at giving the ability to rapidly prototype performance tools
 - Create specialized tools to meet the needs of application developers
- Goal is to recreate Open|SpeedShop from the reusable components
- <http://ft.ornl.gov/doku/cbtw/start>

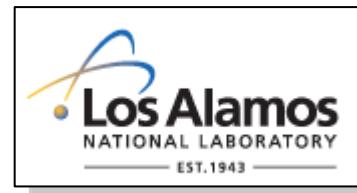
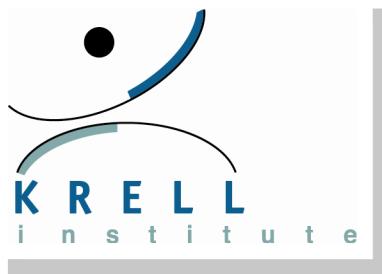
❖ NASA SBIR phase I award

- We are doing a feasibility study & prototyping a new Open|SpeedShop GUI in hopes of winning a phase II award
- <http://openspeedshop.github.com/gui/>

Open | SpeedShop™

Conclusions

SciDAC 2011 Tutorial
How to Analyze the Performance of Parallel Codes 101
A case study with Open|SpeedShop



❖ Where do I spend my time?

- Flat profiles (pcsample)
- Getting inclusive/exclusive timings with callstacks (usertime)
- Identifying hot callpaths (usertime + HP analysis)

❖ How do I analyze cache performance?

- Measure memory performance using hardware counters (hwc)
- Compare to flat profiles (custom comparison)
- Compare multiple hardware counters ($N \times$ hwc, hwcsamp)

❖ How to identify I/O problems?

- Study time spent in I/O routines (io)
- Compare runs under different scenarios (custom comparisons)

❖ How do I find parallel inefficiencies?

- Study time spent in MPI routines (mpi)
- Look for load imbalance (LB view) and outliers (CA view)

❖ Important runtime environment variables (module file)

➤ OPENSS_RAWDATA_DIR

- Shared (across nodes) file system dir where raw data files are written

➤ OPENSS_DB_DIR

- Need a file system with locking enabled: directory path for database
- lustre file system may not have locking enabled

➤ OPENSS_PLUGIN_PATH

- Path to directory where plugins are stored

➤ OPENSS_MPI_IMPLEMENTATION (if multiple)

- If OpenSpeedShop was built with multiple MPI implementations, this points openss at the one you are using in your application
- Also, only required if using the mpi, mpit, or mpotf experiments

➤ LD_LIBRARY_PATH, PATH

- Linux path variables

➤ Example

```
export OPENSS_PREFIX=/home/kg/local
export OPENSS_MPI_IMPLEMENTATION=openmpi
export OPENSS_PLUGIN_PATH=$OPENSS_PREFIX/lib64/openspeedshop
export LD_LIBRARY_PATH=$OPENSS_PREFIX/lib64:$LD_LIBRARY_PATH
export PATH=$OPENSS_PREFIX/bin:$PATH
```

❖ Open|SpeedShop User Guide Documentation

- http://www.openspeedshop.org/docs/user_guide/
- .../share/doc/packages/OpenSpeedShop/users_guide

❖ Python Scripting API Documentation

- http://www.openspeedshop.org/docs/pyscripting_doc/
- .../share/doc/packages/OpenSpeedShop/pyscripting_doc

❖ Command Line Interface Documentation

- http://www.openspeedshop.org/docs/user_guide/
- .../share/doc/packages/OpenSpeedShop/users_guide

❖ Man pages: OpenSpeedShop, osspcsamp, ossmpi, ...

❖ Quick start guide downloadable from web site

- <http://www.openspeedshop.org>
- Click on “Download Quick Start Guide” button

- ❖ **Current version: 2.0.1 (beta) has been released**
- ❖ **Open|SpeedShop Website**
 - <http://www.openspeedshop.org/>
- ❖ **Open|SpeedShop Forum**
 - <http://www.openspeedshop.org/forums/>
- ❖ **Download options:**
 - Package with install script
 - Source for tool and base libraries
- ❖ **Platform Specific Build Information**
 - <http://www.openspeedshop.org/wp/platform-specific-build-information/>
- ❖ **Feedback**
 - Bug tracking available from website
 - oss-questions@openspeedshop.org
 - Feel free to contact presenters directly